THE ESSENCE OF CODATA AND ITS IMPLEMENTATION

by

ZACHARY JOSEPH SULLIVAN

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2018

THESIS APPROVAL PAGE

Student: Zachary Joseph Sullivan

Title: The Essence of Codata and Its Implementation

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Zena Ariola	Chair

and

Sara D. Hodges	Interim Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2018

THESIS ABSTRACT

Zachary Joseph Sullivan

Master of Science

Department of Computer and Information Science

June 2018

Title: The Essence of Codata and Its Implementation

Data types are a widely-used feature of functional programming languages that allow programmers to create abstractions and control branching computations. Instances of data types are introduced by applying one of a disjoint set of constructors and are eliminated by pattern matching on the constructor used. Dually, codata types are defined by their destructors, are introduced by copattern matching on their context, and eliminated by applying destructors.

We extend motivation for codata types to include adding types that satisfy the extensional laws and adding an abstraction for constraining clients of code. We also improve on work implementing codata by developing an untyped compilation technique for codata that works for both call-by-name and call-by-value evaluation strategies and scales to simple and indexed type systems. We demonstrate the practicality of our technique by implementing a prototype compiler and a Haskell language extension.

CURRICULUM VITAE

NAME OF AUTHOR: Zachary Joseph Sullivan

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Indiana University, Bloomington, IN, USA

DEGREES AWARDED:

Master of Science, Computer Science, 2018, University of Oregon
Bachelor of Science, Computer Science, 2016, Indiana University

AREAS OF SPECIAL INTEREST:

Compilation
Programming Languages
Type Systems

PROFESSIONAL EXPERIENCE:

Undergraduate Employee, Indiana University, 2015-2016
Researcher, Indiana University, 2016-2017
Graduate Employee, University of Oregon, 2017-present

ACKNOWLEDGEMENTS

Firstly, I would like to thank my family for their support through my schooling. They have always provided me with a wealth of encouragement.

Academically, several people gave me assistance and encouragement as I grew. I would like to thank my undergraduate professors Amr Sabry and Chung-cheih Shan for helping me at the start of my research. At the University of Oregon, I would like to thank everyone in the programming languages research group. Luke Maurer for helping me through my initial struggles working with compilers specifically GHC. Philip Johnson-Freyd for stimulating my thoughts about programming languages' connections to other areas of study. Paul Downen for advising me through much of this project. And I would especially like to thank my advisor, Zena Ariola who had the patience to help my ramblings become coherent thoughts and for guiding me through my graduate education.

This work is supported by the National Science Foundation under grants CCF-1719158 and CCF-1423617.

TABLE OF CONTENTS

Chapter Page

I. OVERVIEW 1

II. WHAT IS CODATA 3

2.1. Data and Codata: Definition and Use 4

2.2. Strategy Agnostic Code 9

2.3. Indexed Data and Codata 12

III. λ^{COP} : A LANGUAGE WITH NESTED (CO)PATTERNS 17

3.1. Programming in λ^{cop} 17

3.2. Syntax 19

3.3. Type System 22

3.4. Operational Semantics 27

3.5. Type Safety 31

IV. HOW TO COMPILE CODATA TO DATA 32

4.1. Flattening λ^{cop} 33

4.2. A Target Language 36

4.3. Eliminating Codata 37

V. IMPLEMENTATIONS 44

5.1. λ^{cop} Prototype Compiler 44

5.2. Copatterns Haskell Language Extension 49

VI. APPLICATIONS 53

6.1. Programming 53

6.2. Reasoning 62

Chapter Page

VII. DISCUSSION 67

7.1. Comparison with other work 67

7.2. Contributions 68

7.3. Future Work 69

APPENDIX: CODE GENERATION 71

A.1. Haskell 71

A.2. Ocaml 73

A.3. Racket 75

REFERENCES CITED 79

LIST OF FIGURES

Figure Page

1. Syntactic sugar 19
2. Syntax for λ^{cop} 20
3. Well-typed program. 23
4. Type well-formedness judgements 23
5. Judgements for typing terms 24
6. Judgementes for typing (co)alternatives 25
7. Judgements for typing (co)patterns 26
8. Judgements for observable contexts 26
9. Evaluation contexts and values for call-by-name λ^{cop} 27
10. Evaluation contexts and values for call-by-value λ^{cop} 27
11. Parametric operational semantics for λ^{cop} 28
12. Syntax for λ_F^{cop} 34
13. Parametric operational semantics for λ_F^{cop} 35
14. Syntax for λ^{pat} 36
15. Translation from λ_F^{cop} to $\lambda_{\mathcal{N}}^{pat}$ 38
16. Translation from λ_F^{cop} to $\lambda_{\mathcal{V}}^{pat}$ 41

LIST OF TABLES

Table Page

1. fib(40) micro-benchmarks for λ^{cop} backends. 48

2. Micro-benchmarks for GHC implementation. 52

CHAPTER I

OVERVIEW

This thesis develops a notion of codata in programming languages. We divide this task into two major parts. The first, composed of Chapters II, III, and IV, considers the basic usage of data and codata, how the two can be combined in a single programming language, and how codata can compile into data. The second, composed of Chapters V and VI, discusses the integration of codata with current programming language systems and applications.

Chapter II introduces codata by comparing and contrasting it with data. We describe its basic usage and discuss some of the benefits of having codata in a programming language.

In Chapter III, we formalize a programming language with (co)data and nested (co)patterns that we call λ^{cop}. The language emphasizes the duality of matching. We construct a type system for λ^{cop} and give it both a call-by-value and call-by-name operational semantics.

In Chapter IV, we specify a new compilation technique from our source language into a call-by-value and call-by-name target language.

In Chapter V, we describe our implementations of the compilation technique described in Chapter IV. The implementations include a compiler for λ^{cop} that has backends for Racket, Ocaml, and Haskell and a language extension for the Glasgow Haskell Compiler (GHC) that adds codata to the Haskell source language. We give performance figures for our generated code and show how we connect codata to the `IO` monad in applications with the Haskell language extension.

In Chapter VI, we shift our focus to further applications of codata. We give simple examples of how codata can be used in programming scheduling and access

control applications. We end by examining the benefits of codata's equational properties.

In Chapter VII , we discuss related work and conclude with future directions that reiterate the message of this theses: codata is a useful notion and can be easily implemented in current programming languages.

CHAPTER II

WHAT IS CODATA

Functional programmers are familiar with using data types to create abstractions for use in their programs. A simple, practical example of data is a product structure that can be broken into its components. A programmer can use a product to take in or return multiple values. Data types are also introduced by choosing from a disjoint union of constructors. The constructed object can be inspected allowing the computation to branch depending on which constructor was used. A canonical example of using data to control branching is by using the Bool type and specifying one computation if the constructor was True and another if it was False. Thus, data types as a language feature provide a generalized interface for constructing objects with multiple components and branching computations on them. The programmer declares data types at the top level of a program and can use them in the body of the program by introducing instances of the data by applying constructors to arguments and eliminating instances of the data by pattern matching case expressions.

We can invert the notion of a data type to get its dual: a codata type. A codata type describes an object in terms of the observations one can perform on it. Instead of constructing a data structure, we construct a context or build up the observations we want to conduct on the codata structure. The codata structure itself performs copattern matching on these observations.

Describing a structure by their observations or messages reminds one of object-oriented programming. So the natural question to ask is what are the benefits of codata and copattern matching? We will answer this question, but let us first start with presenting some examples of data and codata types.

2.1 Data and Codata: Definition and Use

To describe the usage of codata types, we will compare their declarations, introduction forms, and elimination forms with that of data types. For the following examples, we match data on the left of the pipe with the codata on the right. We will start with the example of a pair which can be expressed both in terms of data and codata:

$$\begin{array}{l} \textbf{data}\ A \times B\ \textbf{where} \\ \quad \mathsf{Pair} : A, B \to A \times B \end{array} \quad \Big| \quad \begin{array}{l} \textbf{codata}\ A \,\&\, B\ \textbf{where} \\ \quad \mathsf{Fst} : A \,\&\, B \to A \\ \quad \mathsf{Snd} : A \,\&\, B \to B \end{array}$$

For the product type $A \times B$, we define only a single *constructor*, Pair, with two arguments. The constructor also serves as a pattern and we require that it is always fully applied. For the codata type $A \,\&\, B$, which we call "with", we have two *destructors* that project out the first and second element. As with constructors serving as patterns for data types, the destructors also serve as copatterns. And while constructors build data, destructors build a form of evaluation contexts which we call *observable*, that is, contexts which can be matched by copatterns. The two declaration forms can be seen as corresponding to the *verificationist* and *pragmatist* approach to inference systems as discussed by Dummett [7]. In the verificationist approach the focus is on the introduction rules (constructors), and the elimination rules (i.e. pattern matching) are justified with respect to the introduction rules. In the pragmatist approach the focus is on the elimination rules (destructors) and the introduction rules are justified (i.e. copattern matching) with respect to the eliminations rules.

To introduce data we only require that we fully apply one of the constructors, whereas to introduce codata we need to specify a computation to handle each destructor.

$$M \equiv (\mathsf{Pair}\ 42\ \mathsf{True}) : \mathbb{Z} \times \mathsf{Bool} \quad \Big| \quad N \equiv \left\{ \begin{array}{c} \mathsf{Fst}\ [\cdot] \to 42 \\ \mathsf{Snd}\ [\cdot] \to \mathsf{True} \end{array} \right\} : \mathbb{Z}\ \&\ \mathsf{Bool}$$

M builds a pair of an integer and a boolean; N uses a less familiar language construct which we refer to as a list of coalternatives. Coalternative lists are called "merge" in Hagino's work [8] and "cofunction" in Regis-Gianas and Laforgue [11]. Each coalternative pairs a copattern with a computation. A copattern describes the shape of an observable context. Thus, the coalternative "$\mathsf{Fst}\ [\cdot] \to 42$" can be read as "when my surrounding context has the shape 'Fst $[\cdot]$' return 42", where "$[\cdot]$" refers to the empty context.

To eliminate a data type, we use a case expression that pattern matches on the different shapes of data. To eliminate a codata type we build a context by applying a destructor.

$$\mathbf{case}\ M\ \{\mathsf{Pair}\ x\ y \to \mathbf{if}\ y\ \mathbf{then}\ x\ \mathbf{else}\ 0\} \quad \Big| \quad \mathbf{if}\ \mathsf{Snd}\ N\ \mathbf{then}\ \mathsf{Fst}\ N\ \mathbf{else}\ 0$$

Elimination of data types in the case expression is where branching can occur. We specify a list of alternatives that pair patterns with computations. In this case, we only have one constructor to match against. On the right-hand side we build the contexts $\mathsf{Snd}\ [\cdot]$ and $\mathsf{Fst}\ [\cdot]$ which can also be seen as sending messages to get the second and first components of the "with" type. Both $\mathsf{Snd}\ [\cdot]$ and $\mathsf{Fst}\ [\cdot]$ are examples of observable evaluation contexts. Not all evaluation contexts are

observable; for instance, **if** $[\cdot]$ **then** $\mathsf{Fst}\ N$ **else** 0 is an evaluation context but it is not observable because we do not have a copattern that matches it.

Another example of codata is the function type. Indeed, a function does not compute till its surrounding context has the shape $[\cdot]\ R$, which we call an *applicative context*. One can see a function definition in terms of copattern matching as follows:

$$\lambda\, x.\, M \triangleq \{[\cdot]\ x \to M\}$$

where "$[\cdot]\ x$" is an applicative copattern. For simplicity, we will often use the more familiar λ-notation.

As in category theory we would like to turn a data declaration into a codata declaration by turning the constructors into destructors and vice-versa. For example, the data declaration which is dual to the "with" type $A \,\&\, B$ is the sum type:

$$\textbf{data}\ A + B\ \textbf{where}$$
$$\mathsf{Left} : A \to A + B$$
$$\mathsf{Right} : B \to A + B$$

Corresponding to the destructors Fst and Snd, we have now two constructors Left and Right. We could do the same with the product type by blindly turning the arrow around:

$$\textbf{codata}\ A \times^d B\ \textbf{where}$$
$$\mathsf{Pair}^d : A \times^d B \to A, B$$

where we superscript with a d the product connective to represent its dual. In functional languages we are accustomed to have multiple assumptions, but what does it mean to have multiple conclusions? Indeed, to capture this we need to step outside functional programming [4, 6] and embrace *effects* such as operators that

modify the flow of control. The dual of the product (also called tensor) is the "par" connective, written $A \parr B$, which allows one to choose between two observable contexts.

2.1.1 Nesting. We have seen how to construct data types like $A \times B$ where A and B are atomic types. However, it is also possible to form a nested product type like $(\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z}$. For example, we can build M of the form $\mathsf{Pair}\ (\mathsf{Pair}\ 2\ 3)\ 4$. To eliminate an instance of this type we need to pattern match twice:

$$\textbf{case}\ M\ \left\{\mathsf{Pair}\ x\ y \to \textbf{case}\ x\ \left\{\mathsf{Pair}\ w\ z \to w + z + y\right\}\right\}$$

We can nest patterns to shrink our code. Instead of having multiple case expressions nested that inspect only one layer of the data structure at a time, we collapse the cases into a single expression with nested patterns.

$$\textbf{case}\ M\ \{\mathsf{Pair}\ (\mathsf{Pair}\ w\ z)\ y \to w + z + y\}$$

In the same manner as patterns, we nest copatterns to match on larger sections of the observable context at once. So instead of writing:

$$\left\{\begin{array}{lcl} \mathsf{Fst}\ [\cdot] & \to & 0 \\ \mathsf{Snd}\ [\cdot] & \to & \left\{\begin{array}{l} \mathsf{Fst}\ [\cdot] \to 20 \\ \mathsf{Snd}\ [\cdot] \to 22 \end{array}\right\} \end{array}\right\} : \mathbb{Z}\ \&\ (\mathbb{Z}\ \&\ \mathbb{Z})$$

we write:

$$\left\{\begin{array}{rcl} \mathsf{Fst}\ [\cdot] & \to & 0 \\ \mathsf{Fst}\ [\mathsf{Snd}\ [\cdot]] & \to & 20 \\ \mathsf{Snd}\ [\mathsf{Snd}\ [\cdot]] & \to & 22 \end{array}\right\}$$

We read the nested copatterns from the inside out starting with the hole $[\cdot]$. The copattern $\mathsf{Fst}\ [\mathsf{Snd}\ [\cdot]]$ can be read as "the observation where the first is requested after the second". A different reading of the copattern is that it matches the context $\mathsf{Fst}\ [\mathsf{Snd}\ [\cdot]]$.

2.1.2 Mixing Patterns and Copatterns. We can also mix pattern and copattern matching. Indeed, that is what we implicitly do when we invoke a function:

$$\left(\lambda\, x.\ \mathbf{case}\ x\ \left\{\begin{array}{l} \mathsf{True} \to 42 \\ \mathsf{False} \to 0 \end{array}\right\}\right)\ \mathsf{True} \triangleq \left\{\begin{array}{l} [\cdot]\ \mathsf{True} \to 42 \\ [\cdot]\ \mathsf{False} \to 0 \end{array}\right\}\ \mathsf{True}$$

Above-left says that first we copattern match the context $[\cdot]\ x$ and then we pattern match on x. Above-right combines these two steps into a single copattern.

We can construct a coalternative expression that will branch on both data and codata by nesting (co)patterns and mixing them together. We see a larger example below that combines both nesting and mixing (co)patterns.

$$\left\{\begin{array}{rcl} \mathsf{Fst}\ [[\cdot]\ \mathsf{True}] & \to & 42 \\ \mathsf{Fst}\ [[\cdot]\ \mathsf{False}] & \to & 5 \\ [\mathsf{Snd}\ [[\cdot]\ \mathsf{True}]]\ x & \to & x + 42 \\ [\mathsf{Snd}\ [[\cdot]\ \mathsf{False}]]\ 0 & \to & 42 \\ [\mathsf{Snd}\ [[\cdot]\ \mathsf{False}]]\ x & \to & -x \end{array}\right\} : \mathsf{Bool} \to \mathbb{Z}\ \&\ (\mathbb{Z} \to \mathbb{Z})$$

This expression first expects an applicative context which we see by the shape of the inner-most copattern (either $[\cdot]$ True or $[\cdot]$ False). Next, the expression pattern matches the argument stored in the calling context which is a Bool. It then creates a "with" type whose first element is an integer. The second element is a function which also expects an integer which can be further analyzed. This example demonstrates the level of expressivity that nested (co)patterns can provide in a single coalternative expression.

We notice a point of asymmetry between data and codata, in addition to not being able to express the dual of the tensor product. Copatterns are expressive enough to contain patterns, but patterns cannot contain copatterns. This means that whereas we can express a function $A \to B$ as codata we cannot express its dual, the subtraction connective $A - B$, which would involve embedding a copattern in a pattern.

2.2 Strategy Agnostic Code

Codata types allow us to define structures independently of evaluation strategy. This property is demonstrated by a program that describes an infinite sequence of zeroes. We will describe the call-by-name, call-by-value, and our agnostic encoding by showing how we write a program that accesses the second element of this sequence.

The call-by-name way to encode such a structure is with an infinite list data type that has one constructor containing the current element and the rest of the list.

$$\textbf{data}\ \mathsf{InfList}\ A\ \textbf{where}$$

$$\mathsf{Cons} : A, \mathsf{InfList}\ A \to \mathsf{InfList}\ A$$

With our data type defined, we can then define "zeroes" recursively. We access elements by pattern matching on the structure recursively. The next element of the structure is only produced when we need it.

$$\begin{array}{l}\textbf{let}\ \text{zeroes} = \mathsf{Cons}\ 0\ \text{zeroes}\ \textbf{in} \\ \quad \textbf{case}\ \text{zeroes} \\ \quad\quad \mathsf{Cons}\ x\ (\mathsf{Cons}\ y\ ys) \to y\end{array}$$

This program will return 0 in call-by-name. Data types in call-by-name languages are often understood as codata because arguments of constructors are only evaluated when observed by pattern matching on or returning them.

On the other hand, if we were to evaluate the above expression in a call-by-value language it would loop forever attempting to construct the value "zeroes". The call-by-value solution is to hide the rest of the computation of the infinite structure behind functions because they are not evaluated until given an argument. This requires that we change the data declaration to the following one.

$$\begin{array}{l}\textbf{data}\ \mathsf{InfList}\ A\ \textbf{where} \\ \quad \mathsf{Cons} : A, (() \to \mathsf{InfList}\ A) \to \mathsf{InfList}\ A\end{array}$$

In addition to changing the type to create the sequence of zeroes, we also need to change how we build and access elements of our infinite structure. We can no longer just nest our patterns because we need to apply the second argument of

Cons to () to compute the next part of the infinite stream.

$$
\begin{array}{l}
\textbf{let}\ \text{zeroes} = \mathsf{Cons}\ 0\ (\lambda\, x.\, \text{zeroes})\ \textbf{in} \\
\quad \textbf{case}\ \text{zeroes} \\
\qquad \mathsf{Cons}\ x\ xs \rightarrow \\
\qquad\quad \textbf{case}\ xs\ () \\
\qquad\qquad \mathsf{Cons}\ y\ ys \rightarrow y
\end{array}
$$

Thus, in order to represent an infinite structure we need to take the evaluation strategy into consideration, and write different programs accordingly. We can avoid this by defining the infinite structure as codata, called Stream, with two destructors: Head gets the current element and Tail observes the next state of the stream.

$$
\begin{array}{l}
\textbf{codata}\ \mathsf{Stream}\ A\ \textbf{where} \\
\quad \mathsf{Head} : \mathsf{Stream}\ A \rightarrow A \\
\quad \mathsf{Tail} : \mathsf{Stream}\ A \rightarrow \mathsf{Stream}\ A
\end{array}
$$

The notion of a stream indeed captures what lazy evaluation is about, however, instead of being relegated in the semantics of the language it becomes apparent in the type. Our previous use of the infinite structure becomes:

$$
\begin{array}{l}
\textbf{let}\ \text{zeroes} = \left\{ \begin{array}{l} \mathsf{Head}\ [\cdot] \rightarrow 0 \\ \mathsf{Tail}\ [\cdot] \rightarrow \text{zeroes} \end{array} \right\}\ \textbf{in} \\
\quad \mathsf{Head}\ (\mathsf{Tail}\ \text{zeroes})
\end{array}
$$

Because the branches of a codata type are not evaluated until it is placed in a context that it can copattern match on, we do not have an infinite loop. Since this is the case for both call-by-name and call-by-value codata, the above program will produce the same result in both strategies. We argue that it should be the job of the compiler to produce different code for call-by-value and call-by-name evaluations, not the programmer.

2.3 Indexed Data and Codata

When we declare a data type, we have in mind how it will be used and have an added notion of how a well-formed instance of the data will be shaped. Indexed data allows the programmer to add formation constraints on how we construct an instance so that we can verify its structure statically. An example of some data for which this applies is a simple expression language with if-statements and addition, which can be representing with the following type:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Expr}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathsf{Plus} & : & \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr} \\
\mathsf{Num} & : & \mathbb{Z} \to \mathsf{Expr} \\
\mathsf{Boolean} & : & \mathsf{Bool} \to \mathsf{Expr} \\
\mathsf{IfThenElse} & : & \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}
\end{array}
\end{array}
$$

When declaring this type, we have in mind an function that evaluates expressions to numbers. For instance, Plus (Num 20) (Num 22) evaluates to 42. However, the data type we declared also allows us to create nonsense expressions such as Plus (Boolean True) (Num 0) that we cannot evaluate. Using type indices, we can specify the constraints that Plus can only add numbers and IfThenElse's first argument must be a boolean. Statically, the compiler can check that these

constraints hold, preventing nonsense expressions. Below, we have added indices to Expr to encode these constraints; we use an underline to denote indices in types.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Expr}\ \underline{T}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathsf{Plus} & : & \mathsf{Expr}\ \underline{\mathbb{Z}} \to \mathsf{Expr}\ \underline{\mathbb{Z}} \to \mathsf{Expr}\ \underline{\mathbb{Z}} \\
\mathsf{Num} & : & \mathbb{Z} \to \mathsf{Expr}\ \underline{\mathbb{Z}} \\
\mathsf{Boolean} & : & \mathsf{Bool} \to \mathsf{Expr}\ \underline{\mathsf{Bool}} \\
\mathsf{IfThenElse} & : & \mathsf{Expr}\ \underline{\mathsf{Bool}} \to \mathsf{Expr}\ \underline{T} \to \mathsf{Expr}\ \underline{T} \to \mathsf{Expr}\ \underline{T}
\end{array}
\end{array}
$$

For the Plus constructor, the indices require that the arguments must have an index of $\mathbb{Z}$ and the only way to introduce an index of $\mathbb{Z}$ is with the Num constructor. Thus, trying to construct Plus (Boolean True) (Num 0) will result in a type error because Boolean True has the type Expr Bool. IfThenElse receives similar constraints.

Dually, with indexed codata we add constraints to the observation of a structure. To understand how this can be beneficial, consider a simplified notion of a server as an example. A server is an infinite object that we can always send messages to and receive responses from.

$$
\begin{array}{l}
\textbf{codata}\ \mathsf{Server}\ A\ \textbf{where} \\
\quad \mathsf{Get} : \mathsf{Server}\ A \to A \\
\quad \mathsf{Post} : \mathsf{Server}\ A \to A \to \mathsf{Server}\ A
\end{array}
$$

A server is parameterized by the type that it inputs and outputs. We have two destructors for the server: a Get message produces an output and a Post message returns a function from some input to the next state of the server. An

example of a server is one that simply returns the last message posted.

$$\begin{array}{l}\textbf{let}\ \text{msg} = \textbf{ref}\ \ \bot\ \ \textbf{in} \\ \text{server} = \left\{\begin{array}{lcl} \mathsf{Get}\ [\cdot] & \to & !\text{msg} \\ [\mathsf{Post}\ [\cdot]]\ s & \to & \text{msg} := s\ ;\ \text{server} \end{array}\right\}\end{array}$$

We represent the server state with a mutable variable "msg". Initially, the server starts with no state, which we represent with $\bot$ which can be thought of as a null-pointer. If the client applies the Post observation and gives the returned function some input, then we recur after updating the server state. If the client applies the Get destructor, then the server just returns what was stored in its state. An example client-server interaction is below.

$$\mathsf{Get}\ (\mathsf{Post}\ (\mathsf{Post}\ \text{server}\ \text{“hi”})\ \text{“bye”}) = \text{“bye”}$$

When the first post is added, the state is set to "hi". Then the hidden state is updated from "hi" to "bye" with another Post message, and when we request the last message by applying Get we receive the string "bye". What do we do if there has been no posted messages and we request the last message?

$$\mathsf{Get}\ \text{server} = \textbf{fail}$$

Where **fail** corresponds to looping forever. The most obvious way to prevent this is to return some dummy message when Get is applied before any Post message, which could be done by returning an optional value or some special string that the client knows is not truly a post. This solution requires the server's client to know the difference between some failure response and a real one. Another solution is

to use the type system to constrain when a client can send a Get message. Below, we show how we encode this constraint with indexed codata. Again, we denote the indices with an underline.

$$
\begin{aligned}
&\textbf{codata}\ \mathsf{Server}\ A\ \underline{T}\ \textbf{where}\\
&\quad \mathsf{Get} : \mathsf{Server}\ A\ \underline{\mathsf{Safe}} \to A\\
&\quad \mathsf{Post} : \mathsf{Server}\ A\ \underline{T} \to A \to \mathsf{Server}\ A\ \underline{\mathsf{Safe}}
\end{aligned}
$$

We add an extra type level tag $\underline{T}$ to our server type that represents whether or not a message has been sent before. The client can only send the Get message to a server which is Safe. Posting a message to the server is the only way to create a Safe server. We also need to update the source code.

$$
\begin{aligned}
&\textbf{let}\ \mathrm{msg} = \textbf{ref}\ \bot\ \textbf{in}\\
&\mathrm{server} = \quad \textbf{let}\ g = \left\{ \begin{array}{lcl} \mathsf{Get}\ [\cdot] & \to & !\mathrm{msg} \\ [\mathsf{Post}\ [\cdot]]\ s & \to & \mathrm{msg} := s\ ;\ g \end{array} \right\}\ \textbf{in}\\
&\qquad\qquad \{[\mathrm{Post}\ [\cdot]]\ s \to \mathrm{msg} := s\ ;\ g\}
\end{aligned}
$$

The new codata has the type Server A $\underline{T}$ where the tag is *not* $\underline{\mathsf{Safe}}$. Inside the body of server, we see that upon receiving its first message the server will execute the server g. The inner server g is a safe server so the client could then apply the Get observation or more Post observations.

The program "Get server" now results in a compile time type error instead of undefined behavior at runtime. Thus, the safe server object through indexed codata encodes how its clients interact with it and the type system enforces this interaction. To reiterate, indexed data constrains the construction of data

allowing the elimination of the data to consider fewer cases. Dually, indexed codata constrains the destruction of codata allowing the introduction of the codata to consider fewer cases.

CHAPTER III

λ^{cop} : A LANGUAGE WITH NESTED (CO)PATTERNS

After seeing some introductory examples demonstrating what codata is, we would like to formalize it into a source language that we call λ^{cop}. The language is designed with several goals in mind: it should emphasize the duality between pattern and copattern matching; it should be easy to program with by allowing (co)patterns to be nested, overlapped, and incomplete; and it should support both call-by-name and call-by-value evaluation strategies.

In Section 3.1, we describe a new construct for copattern matching that is dual to the case expression. In Section 3.2, we give the syntax of λ^{cop}. In Section 3.3, we give a type system that includes judgements for both well-formed types and well-typed terms. We end with an operational semantics for both call-by-name and call-by-value strategies and show that both semantics are safe.

3.1 Programming in λ^{cop}

In the examples presented in the introduction, data types were matched with the explicit case expression, whereas observable contexts were matched implicitly when they contained a codata that could copattern match on them. To regain symmetry with respect to matching, we introduce the *cocase expression* that explicitly pairs a list of coalternatives with the structure it matches on: an *observable context*. Thus, we have the dual matching connectives

$$\textbf{case}\ t\ \{alts\} \quad \Big| \quad \textbf{cocase}\ o\ e$$

where t and e are terms and o is an observable context. Since we have higher order functions, we do not require that e be a list of coalternatives. Dually, we do not require that interrogated term of a case expression t be an applied constructor.

When an observable contexts o meets a list of coalternatives in the cocase expression, the context is copattern matched by the coalternatives and the computation branches. Consider the following program containing only implicit contexts and we will show how we can construct observable contexts and a cocase expression for it.

$$\mathsf{Fst}\ \left(\left(\lambda\, x.\ \left\{\begin{array}{lcl}\mathsf{Fst}\ [\cdot] & \to & x \\ \mathsf{Snd}\ [\cdot] & \to & x+4\end{array}\right\}\right)\ 42\right)$$

From the inside-out, the program first applies a function to 42 building a structure of type $\mathbb{Z}\ \&\ \mathbb{Z}$, then the Fst branch is observed.

The λ-expression is just syntactic sugar for the copattern match that binds x in an applicative context. Thus, we can rewrite the inner coalternative list as:

$$\mathsf{Fst}\ \left(\left\{\begin{array}{lcl}\mathsf{Fst}\ [[\cdot]\ x] & \to & x \\ \mathsf{Snd}\ [[\cdot]\ x] & \to & x+4\end{array}\right\}\ 42\right).$$

The context in which the coalternative list occurs is function application to 42 and then observation of Fst, that is, the context $(\mathsf{Fst}\ [[\cdot]\ 42])$. We can now use the cocase expression to put together this context with the codata object:

$$\mathbf{cocase}\ (\mathsf{Fst}\ [[\cdot]\ 42])\ \left\{\begin{array}{lcl}\mathsf{Fst}\ [[\cdot]\ x] & \to & x \\ \mathsf{Snd}\ [[\cdot]\ x] & \to & x+4\end{array}\right\}$$

As the example shows, observable contexts can be nested like the copatterns that match them. With the context written out explicitly, it is more apparent that

$$
\begin{array}{rcl}
e\ t & \triangleq & \mathbf{cocase}\ ([\cdot]\ t)\ e \\
\mathsf{H}\ t & \triangleq & \mathbf{cocase}\ (\mathsf{H}\ [\cdot])\ t \\
\mathbf{let}\ x = t\ \mathbf{in}\ e & \triangleq & \mathbf{cocase}\ ([\cdot]\ t)\ \{[\cdot]\ x \to e\} \\
\lambda\, x.\, t & \triangleq & \{[\cdot]\ x \to t\} \\
\mathbf{fail} & \triangleq & \mathbf{fix}\, x\ \mathbf{in}\ x
\end{array}
$$

Figure 1. Syntactic sugar

copatterns match on the shapes of contexts in the same way patterns match the shapes of constructed data.

This style of writing out contexts explicitly as in an abstract machine may be unfamiliar to functional programmers and can be tedious. Therefore, we give in Figure 1 some syntactic sugar so that we can still write terms in a λ-calculus style. Also in the syntactic sugar, we represent failure as an infinitely looping fixed-point and we have standard let-expressions.

3.2 Syntax

The full λ^{cop} syntax is presented in Figure 2. In our meta-language, we use a superscript to denote a list, thus S^n describes a list of n elements S. If we use a subscript S_i, then we are selecting the ith element of the list S. A list followed by an arrow in a type $S^n \to T$ denotes a function that takes multiple arguments at once. At the term level, we express applying multiple arguments at once with a superscript such as $f\ e^n$ or subscripts $f\ e_0\ \ldots\ e_n$. We describe appending one object x to the front of the list S with x, S. Lastly, for both types and terms $M[N/\alpha]$ denotes substituting N for α in M. For substitutions on copatterns, $q_0[q_1]$ is short-hand for $q_0[q_1/[\cdot]]$, that is, it describes the copattern q_0 after having substituted q_1 for $[\cdot]$.

At the top level, a program expression encapsulates a term within the scope of a list of (co)data declarations. Data declarations contain a type constructor T

Top level

$$
\begin{array}{rrll}
 & \text{program} & ::= & \mathit{decl}^n;\ t \\
\mathit{decl} \in & \text{declaration} & ::= & \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \\
 & & & \quad \left(\mathsf{K}_i : B_i^m \to \mathsf{T}\ X^n\right)^j \\
 & & \mid & \mathbf{codata}\ \mathsf{U}\ X^n\ \mathbf{where} \\
 & & & \quad \left(\mathsf{H}_i : \mathsf{U}\ X^n \to B_i\right)^j
\end{array}
$$

Types

$$
\begin{array}{rrll}
X, Y, Z \in & \text{type variable} & & \\
A, B, C \in & \text{type} & ::= & X \mid A^+ \mid A^- \\
A^+ \in & \text{positive type} & ::= & \mathsf{T}\ A^n \\
A^- \in & \text{negative type} & ::= & \mathsf{U}\ A^n \mid A \to B
\end{array}
$$

Terms

$$
\begin{array}{rrll}
x, y, z \in & \text{variable} & & \\
e, t, u \in & \text{term} & ::= & x \mid \mathbf{fix}\, x\ \mathbf{in}\ t \\
 & & \mid & \mathsf{K}\ t^n \mid \mathbf{case}\ t\ \{\mathit{alt}^n\} \\
 & & \mid & \{\mathit{coalt}^n\} \mid \mathbf{cocase}\ o\ t \\
 & & & \\
\mathit{alt} \in & \text{alternative} & ::= & p \to e \\
p \in & \text{patterns} & ::= & x \mid \mathsf{K}\ p^n \\
 & & & \\
\mathit{coalt} \in & \text{coalternative} & ::= & q \to e \\
q \in & \text{copatterns} & ::= & [\cdot] \mid \mathsf{H}\ q \mid q\ p \\
 & & & \\
o \in & \text{observable context} & ::= & [\cdot] \mid \mathsf{H}\ o \mid o\ t
\end{array}
$$

Figure 2. Syntax for λ^{cop}

applied to a list of type variables X^n where the type variables are bound within the declaration. Each data declaration contains a list of constructors K that are functions from B^m into the type being defined $\mathsf{T}\ X^n$. Since the number m in B^m refers to the arity of the constructor, when m is 0 the constructor K is nullary (*e.g.* the unit type constructor ()). Codata declarations are similar except that the list of constructors are instead destructors which are functions from $\mathsf{U}\ X^n$ to some type B. Unlike constructors, the arity of each destructor must be one. If we allowed for multiple output, then we would no longer have a functional language.

Types are either a variable bound in a (co)data declaration, a positive type, or a negative type. Positive and negative types refer to whether the structure can be pattern or copattern matched on, respectively. Thus, our notion of type polarity differs slightly from other presentations of codata where positive represents observable values and negative represents computations [1, 13]. For λ^{cop}, positive types must be declared data types, whereas negative types are either declared codata types or the built-in function type $A \ \rightarrow \ B$. We can mix positive and negative types freely. For instance, a product of functions would have the type $(A \rightarrow B) \times (C \rightarrow D)$.

The term language for λ^{cop} contains constructs for introduction and elimination of positive and negative types, along with the standard language features: variables and fix points.

The introduction of a positive type is done with the constructor application $\mathsf{K}\ t^n$, where K is defined in a data declaration and must appear fully applied. Case expressions eliminate data types and are constructed from a term and a list of alternatives. An alternative is composed of a pattern and a term which is run if the branch is matched; the patterns can bind variables that occur free in the right-

hand side of the alternative. We have two pattern forms that are variable and applied constructor, where the latter contains a list of patterns. Patterns are the only mechanism in λ^{cop} for binding variables.

The introduction of a negative type uses a list of coalternatives and the elimination uses cocase expressions. The latter is composed of an observable context and a term[1]. We have three observable contexts and copatterns that match them: "hole", or $[\cdot]$, is the observable context representing the current context which is matched by an identical copattern; $\mathsf{H}\ o$ is the observable context representing a destructor applied to another context which is matched by the copattern $\mathsf{H}\ q$; and, the applicative context $o\ t$ is matched by an applicative copattern $q\ p$ representing function application. The applicative copattern is special because it contains a pattern, thus allowing copatterns to match on data types as well as observable contexts!

In our syntax, we see some of the duality between pattern and copattern matching. Firstly, the introduction forms for both data and codata can exist alone; that is, applied constructors such as $\mathsf{Pair}\ 42\ 0$ and coalternative lists such as $\{\mathsf{Fst}\ [\cdot] \to 42\ ,\ \mathsf{Snd}\ [\cdot] \to 0\}$ can be returned and bound to variables. Conversely, the eliminators for (co)data contain special syntax that cannot exist on its own. For data, the list of alternatives that consumes data can only occur in a case expression. For codata, observable contexts can only occur in a cocase expression.

3.3 Type System

The type system for λ^{cop} contains judgments for well-typed programs, well-formed types, well-typed terms, observable contexts, alternatives, coalternatives, patterns, and copatterns.

[1]The cocase expression is closely related to Hagino's "*merge*", but we provide a special syntax for observable contexts [8]

Program $\boxed{\vdash decl^n; t : A^+}$

$$\frac{decl^n = \Sigma \quad \Sigma \Rightarrow \Gamma \quad \Sigma \vdash A^+ \text{ type} \quad \Gamma \vdash t : A^+}{\vdash decl^n; t : A^+}$$

Figure 3. Well-typed program.

Types $\boxed{\Sigma \vdash A \text{ type}}$

$$\frac{\underset{\ldots}{\textbf{data } \mathsf{T}\ X^n \textbf{ where}} \in \Sigma \quad \Sigma \vdash A \text{ type}}{\Sigma \vdash (\mathsf{T}\ X^n)[A/X_i] \text{ type}}$$

$$\frac{\underset{\ldots}{\textbf{codata } \mathsf{U}\ X^n \textbf{ where}} \in \Sigma \quad \Sigma \vdash A \text{ type}}{\Sigma \vdash (\mathsf{U}\ X^n)[A/X_i] \text{ type}}$$

$$\frac{\Sigma \vdash A \text{ type} \quad \Sigma \vdash B \text{ type}}{\Sigma \vdash A \to B \text{ type}}$$

Figure 4. Type well-formedness judgements

Complete programs in λ^{cop} are checked with the judgement $\boxed{\vdash decl^n;\ t : A^+}$. We require that the term of a complete program is a data type and that its type is well-formed. We may use negative types to construct the final value of a program, but since they cannot be observed, they cannot be the final result [10]. The list of declarations is used as the environment in which we check that types are well-formed. For simplicity, we write the list as Σ. In addition to checking well-formedness, Σ induces an initial mapping Γ, from terms to types, in which we check the type of the program's single term. Γ will include all of the constructors and destructors present in Σ.

Terms $\boxed{\Gamma \vdash t : A}$

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : A}{\Gamma \vdash \mathbf{fix}\, x\ \mathbf{in}\ t : A}$$

$$\frac{(\Gamma \vdash t_i : A_i)^n}{\Gamma, \mathsf{K} : A^n \to \mathsf{T}\ B^m \vdash \mathsf{K}\ t^n : \mathsf{T}\ B^m} \qquad \frac{\Gamma \vdash t : A^+ \quad (\Gamma \mid A^+ \vdash alt : B)^n}{\Gamma \vdash \mathbf{case}\ t\ \{alt^n\} : B}$$

$$\frac{\Gamma \mid B^- \vdash o : A \quad \Gamma \vdash t : B^-}{\Gamma \vdash \mathbf{cocase}\ o\ t : A}$$

$$\frac{(\Gamma \mid A^- \vdash coalt : B)^n \quad n > 0}{\Gamma \vdash \{coalt^n\} : A^-} \qquad \frac{}{\Gamma \vdash \{\} : \mathsf{U}\ A^n}$$

Figure 5. Judgements for typing terms

Well-formed type judgements have the form $\boxed{\Sigma \vdash A\ \text{type}}$. Intuitively, this judgement checks that the type constructors in a type have been declared and that they are fully applied and that all types are only constructed from other well-formed types.

Well-typed term judgements have the form $\boxed{\Gamma \vdash t : A}$. The variable and fix-point rules are standard. The case rule requires that the interrogated term be a positive type, whereas the cocase rule requires that its right-hand side (the interrogator) be a negative type. We have made a distinction between empty coalternative lists and non-empty ones, requiring that the former have a codata type; this restriction is only necessary for our compilation technique, which we will discuss in Chapter IV.

Alternative and coalternative judgements have the form $\boxed{\Gamma \mid A^+ \vdash alt : B}$ and $\boxed{\Gamma \mid A^- \vdash coalt : B}$, respectively. The distinguished type to the left of the turnstile is that of the structure being matched, whereas type on the far

Alternative $\boxed{\Gamma \mid A^+ \vdash alt : B}$

$$\frac{\Gamma' \vdash_{pat} p : A^+ \quad \Gamma, \Gamma' \vdash e : B}{\Gamma \mid A^+ \vdash p \to e : B}$$

Coalternative $\boxed{\Gamma \mid A^- \vdash coalt : B}$

$$\frac{\Gamma' \mid A^- \vdash_{cop} q : B \quad \Gamma, \Gamma' \vdash e : B}{\Gamma \mid A^- \vdash q \to e : B}$$

Figure 6. Judgementes for typing (co)alternatives

right is the type being returned. Another reading of the (co)alternative rule is "in the environment Γ the (co)alternative (co)*alt* matches some (co)data of type A (positive for data and negative for codata) and returns a type B". For both alternatives and coalternatives, the (co)pattern match gives us an extended environment in which we check the type of the right-hand side. The two judgements are also dual to one another: alternatives match positive types and coalternatives match negative types.

Pattern and copattern typing judgements have the form $\boxed{\Gamma \vdash_{pat} p : A}$ and $\boxed{\Gamma \mid A^- \vdash_{cop} q : B}$, respectively. Because they represent the same shapes as the structures they match, patterns have rules similar to variables and constructors, whereas copatterns have rules similar to observable contexts. The pattern judgement guarantees that the constructor pattern is fully applied by checking the type of the constructor in the context Γ. The constructor pattern rule also guarantees that the environment is linear as noted by $\Gamma_0, \Gamma_1, \ldots, \Gamma_j$. A linear context restriction is respected by the applicative copattern rule as well, which is necessary because applicative copatterns contain patterns. The pattern and

Patterns $\boxed{\Gamma \vdash_{pat} p : A}$

$$\frac{}{x : A \vdash_{pat} x : A}$$

$$\frac{(\Gamma_j \vdash_{pat} p_j : A_j)^n}{\Gamma_0, \Gamma_1, \ldots, \Gamma_n, \mathsf{K} : A^n \to \mathsf{T}\ B^m \vdash_{pat} \mathsf{K}\ p^n : \mathsf{T}\ B^m}$$

Copatterns $\boxed{\Gamma \mid A^- \vdash_{cop} q : B}$

$$\frac{}{\Gamma \mid A^- \vdash_{cop} [\cdot] : A^-} \qquad \frac{\Gamma \mid A^- \vdash_{cop} q : B \to C \quad \Gamma' \vdash_{pat} p : B}{\Gamma, \Gamma' \mid A^- \vdash_{cop} q\ p : C}$$

$$\frac{\Gamma \mid A^- \vdash_{cop} q : \mathsf{U}\ X^n}{\Gamma, \mathsf{H} : \mathsf{U}\ X^n \to B \mid A^- \vdash_{cop} \mathsf{H}\ q : B}$$

Figure 7. Judgements for typing (co)patterns

copattern rules do not appear dual to each other because copatterns can contain patterns but patterns cannot contain copatterns.

Well-typed observable contexts $\boxed{\Gamma \mid A^- \vdash o : B}$ represent a context with a negative type A^- with an output type B. In the same way the pattern rules are similar to variable and constructor application rules, the observable context

Observable Contexts $\boxed{\Gamma \mid A^- \vdash o : B}$

$$\frac{}{\Gamma \mid A^- \vdash [\cdot] : A^-} \qquad \frac{\Gamma \mid A^- \vdash o : B \to C \quad \Gamma \vdash t : B}{\Gamma \mid A^- \vdash o\ t : C}$$

$$\frac{\Gamma \mid A^- \vdash o : \mathsf{U}\ X^n}{\Gamma, \mathsf{H} : \mathsf{U}\ X^n \to B \mid A^- \vdash \mathsf{H}\ q : B}$$

Figure 8. Judgements for observable contexts

$$
\begin{array}{lll}
V \in & \mathit{values} & ::= t \\
E \in & \mathit{eval\ ctx} & ::= \Box \mid \mathbf{case}\ E\ \{alt^n\} \mid \mathbf{cocase}\ o\ E
\end{array}
$$

Figure 9. Evaluation contexts and values for call-by-name λ^{cop}

$$
\begin{array}{lll}
V \in & \mathit{values} & ::= \mathsf{K}\ V^n \mid \{[\cdot]\ x \to t\} \mid \{\mathsf{H}\ [\cdot] \to t\ ,\ \_ \to u\} \mid \{\} \\
E \in & \mathit{eval\ ctx} & ::= \Box \mid \mathsf{K}\ E^n \mid \mathbf{case}\ E\ \{alt^n\} \\
 & & \quad \mid \mathbf{cocase}\ o\ E \mid \mathbf{cocase}\ ([\cdot]\ E)\ V
\end{array}
$$

Figure 10. Evaluation contexts and values for call-by-value λ^{cop}

rules are similar to the copattern rules. However we are not binding variables in observable contexts, and thus, we do not require that the environment is linear.

3.4 Operational Semantics

The semantics for λ^{cop} are a union of distinct groups of reduction rules: standard rules, matching rules, and flattening rules denoted ($\mapsto$), ($\mapsto_M$), and ($\mapsto_F$), respectively. We specify both a call-by-value and call-by-name operational semantics in a manner similar to other strategy-parametric languages such $\mu\tilde{\mu}$ specified by Curien and Herbelin [4] and λlet from Downen [5]. That is, for each strategy we have a different set of values and evaluation contexts.

In the call-by-name operational semantics, the set of values contains all terms, and there are only three evaluation contexts: the hole, the argument of a case expression, and the right-hand-side of a cocase expression (Figure 9). In the call-by-value version of the operational semantics, the values differ in that constructors are only applied to values, see Figure 10. To meet this added constraint on values, there are additional evaluation contexts to evaluate the arguments of constructors and the top of call-stack in an applicative context.

In Figure 11, we give the parametric operational semantics. Notice that we only substitute values. The two standard rules ($\mapsto$) are for evaluating terms in

$$
\begin{array}{rcl}
E[t] & \mapsto & E[t'] \quad \text{where } t \mapsto t' \\
\mathbf{fix}\, x\ \mathbf{in}\ t & \mapsto & t[\mathbf{fix}\, x\ \mathbf{in}\ t/x]
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{cocase}\ o\ \{\} & \mapsto_M & \mathbf{fail} \\
\mathbf{cocase}\ ([\cdot]\ V)\ \{[\cdot]\ x \to t\} & \mapsto_M & t[V/x] \\
\mathbf{cocase}\ (\mathsf{H}_i\ [\cdot])\ \{\mathsf{H}_i\ [\cdot] \to t\ ,\ \_ \to u\} & \mapsto_M & t \\
\mathbf{cocase}\ (\mathsf{H}_i\ [\cdot])\ \{\mathsf{H}_j\ [\cdot] \to t\ ,\ \_ \to u\} & \mapsto_M & u \\
\mathbf{case}\ \mathsf{K}_i\ V^n\ \{\} & \mapsto_M & \mathbf{fail} \\
\mathbf{case}\ \mathsf{K}_i\ V^n\ \{\mathsf{K}_i\ x^m \to u\ ,\ y \to e\} & \mapsto_M & u[V_i/x_i]^n \\
\mathbf{case}\ \mathsf{K}_i\ V^n)\ \{\mathsf{K}_j\ x^m \to u\ ,\ y \to e\} & \mapsto_M & e[\mathsf{K}_i\ V^n/y]
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{cocase}\ o[o'/[\cdot]]\ t & \mapsto_F & \mathbf{cocase}\ o\ (\mathbf{cocase}\ o'\ t) \\
 & & \quad \text{where } o' \neq [\cdot] \\
\mathbf{cocase}\ [\cdot]\ t & \mapsto_F & t \\
\{[\cdot]\ p \to t\ ,\ coalt^n\} & \mapsto_F & \left\{[\cdot]\ x \to \mathbf{case}\ x\ \begin{Bmatrix} p & \to & t \\ y & \to & \{coalt^n\}\ x \end{Bmatrix}\right\} \\
 & & \quad \text{where } x, y \notin FV(\{coalt^n\}) \\
 & & \qquad\ x \notin FV(t) \\
 & & \qquad\ coalt^n \neq [\cdot] \to u \\
\{\mathsf{H}_i\ [\cdot] \to t\ ,\ coalt^n\} & \mapsto_F & \begin{Bmatrix} \mathsf{H}_i\ [\cdot] & \to & t \\ \_ & \to & \{coalt^n\} \end{Bmatrix} \\
 & & \quad \text{where } coalt^n \neq [\cdot] \to u \\
\{q[[\cdot]\ p] \to t\ ,\ coalt^n\} & \mapsto_F & \left\{\begin{array}{ccc} [\cdot]\ p & \to & \begin{Bmatrix} q & \to & t \\ \_ & \to & \{coalt^n\}\ p \end{Bmatrix} \\ \_ & \to & \{coalt^n\} \end{array}\right\} \\
\{q[\mathsf{H}_i\ [\cdot]] \to t\ ,\ coalt^n\} & \mapsto_F & \left\{\begin{array}{ccc} \mathsf{H}_i\ [\cdot] & \to & \begin{Bmatrix} q & \to & t \\ \_ & \to & \mathsf{H}_i\ \{coalt^n\} \end{Bmatrix} \\ \_ & \to & \{coalt^n\} \end{array}\right\} \\
\mathbf{case}\ t\ \{x \to u\ ,\ alt^n\} & \mapsto_F & u[t/x] \\
\mathbf{case}\ t\ \{\mathsf{K}_i\ x^n \to u\ ,\ alt^m\} & \mapsto_F & \mathbf{case}\ t\ \begin{Bmatrix} \mathsf{K}_i\ x^n & \to & u \\ y & \to & \mathbf{case}\ y\ \{alt^m\} \end{Bmatrix} \\
 & & \quad \text{where } y \notin FV(alt^m) \\
\mathbf{case}\ t\ \{\mathsf{K}_i\ x^n, p_i, p^m \to u\ ,\ alt^k\} & \mapsto_F & \mathbf{case}\ t\ \begin{Bmatrix} \mathsf{K}_i\ x^n, x_i, p^m & \to & \mathbf{case}\ x_i\ \{p_i \to u\} \\ y & \to & \mathbf{case}\ y\ \{alt^k\} \end{Bmatrix} \\
 & & \quad \text{where } x_i \notin FV(u) \wedge y \notin FV(alt^k)
\end{array}
$$

Figure 11. Parametric operational semantics for λ^{cop}

evaluation contexts and the fix expression. When a term is a case expression with an applied constructor or a cocase expression with a list of coalternatives on the right-hand side, then the next step is with one of the matching rules. If the set of (co)alternatives is empty, then the (co)case expression will result in failure. If the constructor or destructor matches the one in the pattern or copattern, then we take the first branch which is for success, otherwise we take the second branch as a fall through branch. The matching rules are not entirely symmetric. Patterns can bind variables, so the reduction rules contain substitutions. Also copattern matching has one more rule than pattern matching that matches an applicative context. This rule will always succeed since it contains the variable pattern only.

3.4.1 Flattening (Co)patterns and (Co)alternatives. The set of flattening rules are the same in both call-by-name and call-by-value strategies. These rules are required to use any of the matching rules because they only match flattened versions of (co)alternative lists. The flattening rules were greatly inspired by Augustsson's work on compiling patterns [3]. Here, we will only focus on flattening copatterns, observable contexts, and coalternatives. Intuitively, the rules flatten all of these new negative constructs such that only one part of the structure is matched at a time.

To flattening copatterns, we construct a new coalternative where the first branch checks the inner-most copattern then checks the rest of the copattern. We also create the default branch which contains the rest of the coalternatives. For example,

$$\{\mathsf{Fst}\ [\mathsf{Snd}\ [\cdot]] \to 42\} \mapsto_F \left\{ \begin{array}{ccc} \mathsf{Snd}\ [\cdot] & \to & \{\mathsf{Fst}\ [\cdot] \to 42\} \\ \_ & \to & \{\} \end{array} \right\}$$

For the newly flattened copatterns to match observable contexts we also need to flatten nested observable contexts. This is done by pushing the inner-most contexts into the new cocase expressions. For example,

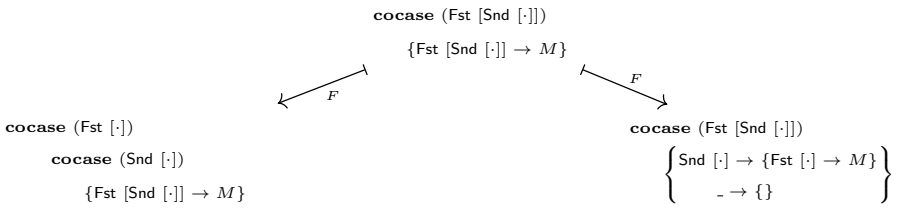
$$\textbf{cocase}\ (\mathsf{Fst}\ [\mathsf{Snd}\ [\cdot]])\ N \mapsto_F \textbf{cocase}\ (\mathsf{Fst}\ [\cdot])\ (\textbf{cocase}\ (\mathsf{Snd}\ [\cdot])\ N)$$

Finally, we have a rule for removing the useless observable context containing only the current context $[\cdot]$.

(Co)alternative lists are flattened so that there are only two branches left: a main branch which checks part of the structure and a default branch that contains the rest of list.

$$\left\{\begin{array}{lcl} \mathsf{Fst}\ [\cdot] & \to & R \\ \mathsf{Snd}\ [\cdot] & \to & S \end{array}\right\} \mapsto_F \left\{\begin{array}{lcl} \mathsf{Fst}\ [\cdot] & \to & R \\ \_ & \to & \{\mathsf{Snd}\ [\cdot] \to S\} \end{array}\right\}$$

There are different orders in which we can apply the flattening rules to evaluate a term in λ^{cop}, as shown in the example below.



In a more complicated term there will be even more ways that we can choose to flatten. However, since the flattening rules are confluent the order will not impact the end result.

3.5 Type Safety

Both the call-by-value and call-by-name operational semantics are safe in λ^{cop}'s type system. Here we sketch some of the notable details of the proof. In the proof, we consider both call-by-name and call-by-value at the same time because in many cases the behavior of the semantics is the same for both strategies.

Our theorem of progress for type safety will only work on closed terms, but we need to have constructors and destructors in scope to give data and codata their types. Thus, we consider progress for entire programs instead of terms. Also recall that **fail** maps to $\mathbf{fix}\, x\ \mathbf{in}\ x$, so we will not get stuck on these terms instead we will make "progress" forever.

Theorem 1 (Progress)**.** *For both the call-by-name and call-by-value operational semantics, if* $\vdash \mathit{decl}^n; t : A^+$*, then* t *is either a value or there exists some* t' *such that* $t \mapsto t'$*.*

Proof. By induction on the derivation of $\vdash \mathit{decl}^n; t : A$. □

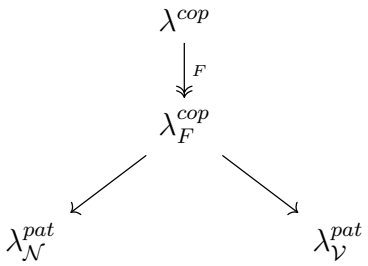
Theorem 2 (Preservation)**.** *If* $\Gamma \vdash t : A$ *and there exists* t' *such that* $t \mapsto t'$*, then* $\Gamma \vdash t' : A$*.*

Proof. By induction on the derivation $\Gamma \vdash t : A$. □

CHAPTER IV

HOW TO COMPILE CODATA TO DATA

Having specified the full syntax of λ^{cop} and given some basic examples of codata usage, we now consider how to augment current functional programming languages with codata and copatterns. We will translate λ^{cop} into a common functional language by first distilling a computational core and then reducing codata into a target language's data and function types. This requires a slightly different method for each evaluation strategy because of their different behavior in the case of data types. The diagram below depicts the full compilation pipeline that we present.



This chapter is separated into three sections. In Section 4.1, we remove nested patterns and copatterns, introducing a flattened sub-language λ_F^{cop}. In Section 4.2 we specify the target language of our translation: λ^{pat}, which represents a simple functional language with data types. Finally in Section 4.3, we give a translation from λ_F^{cop} to both a call-by-name and call-by-value version of λ^{pat}.

As an example throughout the chapter, we will trace the compilation of the following program that observes the first element of a stream of zeroes.

$$\begin{array}{c}\textbf{codata}\ \mathsf{Stream}\ A\ \textbf{where}\\ \mathsf{Head} : \mathsf{Stream}\ A \to A\\ \mathsf{Tail} : \mathsf{Stream}\ A \to \mathsf{Stream}\ A\end{array}$$

$$\textbf{cocase}\ (\mathsf{Head}\ [\cdot])\ \left(\textbf{fix}\, s\ \textbf{in}\ \left\{\begin{array}{c}\mathsf{Head}\ [\cdot] \to 0\\ \mathsf{Tail}\ [\cdot] \to s\end{array}\right\}\right)$$

4.1 Flattening λ^{cop}

In order to simplify our final compilation step, we distill a core language from λ^{cop} language that maintains all the expressiveness of the source. In Chapter II, we saw that nesting offers expressive flexibility for both pattern and copattern matching. In the operational semantics given in Chapter III, we saw that the matching rules eliminate evaluation contexts and perform substitution. Thus they are the core computational rules for λ^{cop}. We also saw that the matching reductions can only occur on a (co)pattern with the form x, $\mathsf{K}\ x^n$, $\mathsf{H}\ [\cdot]$, or $[\cdot]\ x$. These are referred to as *flat (co)patterns*. Since we rely only on flat (co)patterns to compute, we can simplify our language by using only flat (co)patterns and preserve completeness.

The flattening subset of the operational semantics ($\mapsto_F$) is used to unnest (co)patterns, to unnest observable contexts (so that they match flat patterns), and to turn lists of (co)alternatives into just success and failure branches. We can use these rules as rewriting rules ($\to_F$), where the reflexive, transitive closure

Terms

$$
\begin{array}{rrl}
x, y, z \in & \text{variable} & \\
e, t, u \in & \text{term} ::= & x \mid e\ t \mid \mathbf{fix}\, x\ \mathbf{in}\ t \\
& \mid & \mathsf{K}\ t^n \mid \mathbf{case}\ t\ \{p \to e\ ,\ y \to u\} \\
& \mid & \mathsf{H}\ t \mid \mathbf{cocase}\ o\ t \mid \{q \to e\ ,\ \_ \to u\} \mid \{\} \\
\\
p \in & \text{patterns} ::= & x \mid \mathsf{K}\ x^n \\
q \in & \text{copatterns} ::= & \mathsf{H}\ [\cdot] \mid [\cdot]\ x \\
o \in & \text{observable context} ::= & \mathsf{H}\ [\cdot] \mid [\cdot]\ t
\end{array}
$$

Figure 12. Syntax for λ_F^{cop}

produces terms only containing flattened (co)patterns. We call this sub-syntax λ_F^{cop}, see Figure 12. The flattened syntax has a non-recursive set of (co)patterns and observable contexts and a list of (co)alternatives with a maximum length of 2.

After the flattening pass our example program becomes the following:

$$
\mathbf{cocase}\ (\text{Head}\ [\cdot])\ \left(\mathbf{fix}\, x\ \mathbf{in}\ \left\{\begin{array}{lcl} \text{Head}\ [\cdot] & \to & 0 \\ \_ & \to & \left\{\begin{array}{lcl} \text{Tail}\ [\cdot] & \to & x \\ \_ & \to & \{\} \end{array}\right\} \end{array}\right\}\right)
$$

Since there are no nested (co)patterns or observable contexts, only the list of coalternatives is changed to a new list containing only two options. The last branch that can be checked —after first attempting to match the context Head then Tail— is the empty list of coalternatives meaning we will fail if we take that branch. This final default branch will never be evaluated because our example completely covers the type Stream $\mathbb{Z}$ with its coalternatives; however, we still require this unused branch for compiling to typed languages as we will see in Section 4.3.

After obtaining a term in the sub-syntax, we can simplify the operational semantics since we no longer need to flatten during evaluation. In Figure 13, we

$$
\begin{array}{rcll}
E[t] & \mapsto & E[t'] & \text{where } t \mapsto t' \\
\mathbf{fix}\, x\ \mathbf{in}\ t & \mapsto & t[\mathbf{fix}\, x\ \mathbf{in}\ t/x] & \\
 & & & \\
\mathbf{cocase}\ o\ \{\} & \mapsto_M & \mathbf{fail} & \\
\mathbf{cocase}\ ([\cdot]\ V)\ \{[\cdot]\ x \to t\} & \mapsto_M & t[V/x] & \\
\mathbf{cocase}\ (\mathsf{H}_i\ [\cdot])\ \{\mathsf{H}_i\ [\cdot] \to t\ ,\ \_ \to u\} & \mapsto_M & t & \\
\mathbf{cocase}\ (\mathsf{H}_i\ [\cdot])\ \{\mathsf{H}_j\ [\cdot] \to t\ ,\ \_ \to u\} & \mapsto_M & u & \\
\mathbf{case}\ \mathsf{K}_i\ V^n\ \{\} & \mapsto_M & \mathbf{fail} & \\
\mathbf{case}\ \mathsf{K}_i\ V^n\ \{\mathsf{K}_i\ x^n \to u\ ,\ y \to e\} & \mapsto_M & u[V_i/x_i]^n & \\
\mathbf{case}\ \mathsf{K}_i\ V^n\ \{\mathsf{K}_j\ x^m \to u\ ,\ y \to e\} & \mapsto_M & e[\mathsf{K}_i\ V^n/y] &
\end{array}
$$

Figure 13. Parametric operational semantics for λ_F^{cop}

show the reduced rules required to evaluate the sub-syntax of λ_F^{cop}. As before the semantics is parametric with respect to the definition of evaluation context E and value V.

To be certain that we can consider only the sub-syntax in our final compilation step, we must show that our sub-syntax is closed under the total set of reduction rules. If that is the case, then it is impossible to create a non-flattened term during evaluation.

Theorem 3 (Closed under $(\mapsto\!\!\!\!\rightarrow)$)**.** *If $t \in \lambda_F^{cop}$ and $t \mapsto\!\!\!\!\rightarrow t'$ then $t' \in \lambda_F^{cop}$.*

Next we want to consider properties that should be considered when we are using flattening as a compilation step. Firstly, flattening should be strongly normalizing so that we can always finish compiling. Secondly, flattening should be confluent so that we can apply the rules anywhere and not change the meaning of the program, which we already discussed in Chapter III. Lastly, flattening should commute with the total set of reduction rules so that we can choose to flatten at any time.

Theorem 4 (Strong Normalization of $\twoheadrightarrow_F$)**.** *All reduction sequences $t \twoheadrightarrow_F t'$ are finite.*

Theorem 5 (Communitivity)**.** *If $t \twoheadrightarrow_F t_1$ and $t \mapsto\!\!\!\!\rightarrow t_2$, then there exists some u such that $t_1 \mapsto\!\!\!\!\rightarrow u$ and $t_2 \twoheadrightarrow_F u$.*

Top level

$$\begin{array}{rrl} & \text{program} ::= & decl^n;\ t \\ decl \in & \text{declaration} ::= & \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \\ & & \quad \left(\mathsf{K}_i : B_i^m \to \mathsf{T}\ X^n\right)^j \end{array}$$

Types

$$\begin{array}{rrl} X, Y, Z \in & \text{type variable} & \\ A, B, C \in & \text{type} ::= & X \mid \mathsf{T}\ A^n \mid A \to B \end{array}$$

Terms

$$\begin{array}{rrl} x, y, z \in & \text{variable} & \\ e, t, u \in & \text{term} ::= & x \mid \mathbf{fix}\, x\ \mathbf{in}\ t \\ & \mid & \mathsf{K}\ t^n \mid \mathbf{case}\ t\ \{alt^n\} \\ & \mid & \lambda\, x.\, t \mid e\ t \\ \\ alt \in & \text{alternative} ::= & p \to e \\ p \in & \text{patterns} ::= & x \mid \mathsf{K}\ x^n \end{array}$$

Figure 14. Syntax for λ^{pat}

4.2 A Target Language

The target language for our translation should meet two requirements. There should be some built-in codata or observation consuming expression that we will map all of our codata onto. We use λ-expressions for this because they are wide spread in programming languages. We do not require data types (since they too can be encoded with λ-expressions), but we will use them for simplicity. The second requirement is that if the target language has a type system, then we require that it supports polymorphism. This is a finer detail that will be discussed more in the final translation.

We call our target language λ^{pat} and specify its syntax in Figure 14. We return to the applicative forms and function introductions from λ-calculus. For types, we have data types and one built-in negative type: functions. We only have

flat patterns because λ_F^{cop} only has flat patterns, but there is nothing preventing the target language from being more expressive than the one we specify here. And lastly, we still have general recursion in the language with the fix expression.

Because λ^{pat} is a sub-syntax of λ^{cop} and its operational rules are a subset of $(\mapsto)$, λ^{pat} could be regarded as another sub-language of λ^{cop} along with λ_F^{cop}. The operational semantics for λ^{pat} is even smaller than the flattened language since it only contains the matching rules for case and applicative cocase from λ^{cop}.

4.3 Eliminating Codata

The final step in implementing λ^{cop} is the elimination of codata. This requires that we encode the branches specified in coalternative lists with data. Because the branches represent computations, our translation should guarantee that the branches are not run until a translated destructor is applied to our object.

Thus far, both call-by-name and call-by-value strategies have received similar treatment. The source syntax λ^{cop} and the flattening rules are the same for both evaluation strategy. In the operational semantics, the strategies only differ in constructor and function application since the notion of value changes for call-by-value and call-by-name. When we compile codata to data, we need to provide a different method for each strategy because they provide different mechanisms for controlling when a computation occurs. As we saw in the introduction, we need to explicitly thunk expressions in call-by-value to control computations, whereas call-by-name computations only occur when they are called or matched on. We will describe the call-by-name translation first because it is simpler.

To give a high-level overview of how we compile away codata, we represent the coalternatives for a given codata type by a data type with a single constructor that has arguments for holding computations for each destructor of that codata. To

Declarations

$$
\begin{array}{rcl}
[\![\ \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \quad (\mathsf{K}_i : B_i^m \to \mathsf{T}\ X^n)^j \ ]\!] & \triangleq & \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \quad (\mathsf{K}_i : B_i^m \to \mathsf{T}\ X^n)^j \\
[\![\ \mathbf{codata}\ \mathsf{U}_h\ X^n\ \mathbf{where} \quad (\mathsf{H}_i : \mathsf{U}_h\ X^n \to B_i)^j \ ]\!] & \triangleq & \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \quad \mathsf{K}_h : (\mathsf{Option}\ B_i)^j \to \mathsf{T}\ X^n
\end{array}
$$

Terms

$$
\begin{array}{rcl}
[\![x]\!]_d & \triangleq & x \\
[\![\mathbf{fix}\, x\ \mathbf{in}\ t]\!]_d & \triangleq & \mathbf{fix}\, x\ \mathbf{in}\ [\![t]\!]_d \\
[\![\mathsf{K}\ t^n]\!]_d & \triangleq & \mathsf{K}\ [\![t]\!]_d^n \\
[\![\mathbf{case}\ t\ \{(p \to u)^n\}]\!]_d & \triangleq & \mathbf{case}\ [\![t]\!]_d\ \{(p \to [\![u]\!]_d)^n\} \\
[\![\mathbf{cocase}\ ([\cdot]\ e)\ t]\!]_d & \triangleq & [\![t]\!]_d\ [\![e]\!]_d \\
[\![\mathbf{cocase}\ (\mathsf{H}_i\ [\cdot])\ t]\!]_d & \triangleq & \mathbf{case}\ [\![t]\!]_d \left\{ \begin{array}{ccc} \mathsf{None} & \to & \mathbf{fail} \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ \mathsf{None}_i\ \ldots\ x_n) & \to & \mathbf{fail} \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ (\mathsf{Some}\ y)_i\ \ldots\ x_n) & \to & y \end{array} \right\} \\
 & & \quad \text{where } \mathsf{H}_i \mapsto_d K \\
[\![\{[\cdot]\ x \to t\}]\!]_d & \triangleq & \lambda x.\, [\![t]\!]_d \\
[\![\{\mathsf{H}_i\ [\cdot] \to t\ ,\ \_ \to u\}]\!]_d & \triangleq & \mathbf{case}\ [\![u]\!]_d \left\{ \begin{array}{c} \mathsf{None} \to \\ \mathsf{Some}\ (\mathsf{K}\ \mathsf{None}_0\ \ldots\ (\mathsf{Some}\ [\![t]\!]_d)_i\ \ldots\ \mathsf{None}_n) \\ \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ x_i\ \ldots\ x_n) \to \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ (\mathsf{Some}\ [\![t]\!]_d)_i\ \ldots\ x_n) \end{array} \right\} \\
 & & \quad \text{where } \mathsf{H}_i \mapsto_d K \\
[\![\{\}]\!]_d & \triangleq & \mathsf{None}
\end{array}
$$

Figure 15. Translation from λ_F^{cop} to $\lambda_{\mathcal{N}}^{pat}$

translate the application of a destructor, we match on the translated codata, pull out the branch of that destructor, and run it.

4.3.1 Call-by-name. In Figure 15, we specify the call-by-name translation for both declarations and terms. For data declarations, the translation is the identity function. For codata declarations, we must construct a new data type with a single constructor where there is an argument for each destructor of the codata. We need to wrap each branch in an Option type because we allow the programmer to specify incomplete matches. A branch will be None if it was not specified in the source program.

Consider the following translation of the stream codata type.

$$\begin{aligned}&\textbf{codata}\ \mathsf{Stream}\ A\ \textbf{where}\\ &\quad \mathsf{Head} : \mathsf{Stream}\ A \to A\\ &\quad \mathsf{Tail} : \mathsf{Stream}\ A \to \mathsf{Stream}\ A\end{aligned}$$

$$\begin{aligned}&\textbf{data}\ \mathsf{Stream}\ A\ \textbf{where}\\ &\quad \mathsf{MkS} : \mathsf{Option}\ A, \mathsf{Option}\ (\mathsf{Stream}\ A) \to \mathsf{Stream}\ A\end{aligned}$$

We introduce a new data type with the same name and one constructor that pairs together computations representing both destructors of the stream type. The first argument of MkS represents the Head destructor and the second represents the Tail.

At the term level, the translation is parameterized by a map d from destructors to a data constructor created by translating the declarations. The map is described by the subscripts h on the codata types. For our running example we have the map

$$d = \{\mathsf{Head} \mapsto \mathsf{MkS}, \mathsf{Tail} \mapsto \mathsf{MkS}\}.$$

The example term translates as follows.

$$\textbf{cocase}\ (\text{Head}\ [\cdot])\ \left(\textbf{fix}\, x\ \textbf{in}\ \left\{\begin{array}{c}\text{Head}\ [\cdot] \to 0\\ \_ \to \left\{\begin{array}{c}\text{Tail}\ [\cdot] \to x\\ \_ \to \{\}\end{array}\right\}\end{array}\right\}\right)$$

$$\textbf{case}\ \left(\textbf{fix}\, s\ \textbf{in}\ \left(\begin{array}{l}\textbf{case}\ \left(\begin{array}{l}\textbf{case}\ \mathsf{None}\\ \quad \mathsf{None} \to \mathsf{Some}\ (\mathsf{MkS}\ \mathsf{None}\ (\mathsf{Some}\ s))\\ \quad \mathsf{Some}\ (\mathsf{MkS}\ x_0\ x_1) \to \mathsf{Some}\ (\mathsf{MkS}\ x_0\ (\mathsf{Some}\ s))\end{array}\right)\\ \quad \mathsf{None} \to \mathsf{Some}\ (\mathsf{MkS}\ (\mathsf{Some}\ 0)\ \mathsf{None})\\ \quad \mathsf{Some}\ (\mathsf{MkS}\ x_0\ x_1) \to \mathsf{Some}\ (\mathsf{MkS}\ (\mathsf{Some}\ 0)\ x_1)\end{array}\right)\right)$$

$$\begin{array}{l}\mathsf{None} \to \textbf{fail}\\ \mathsf{Some}\ (\mathsf{MkS}\ \mathsf{None}\ x_1) \to \textbf{fail}\\ \mathsf{Some}\ (\mathsf{MkS}\ (\mathsf{Some}\ x_0)\ x_1) \to x_0\end{array}$$

The translation follows the nested structure of flattened coalternatives in the original term. The stream data type begins with the inner most failure branch, setting it to None. Then branches are gradually added by casing on their inner branches and adding computations as new branches. We wrap the whole, translated codata structure in an Option type because we can give the empty list of coalternatives as a valid instance of any codata type. Finally, we apply our observation Head by doing a final case and performing the computation in the desired first branch.

Not shown in the above example is the translation of a function type. Variables, fix-points, applied constructors, and case expressions are all translated by maintaining their structure and applying the translation to sub-terms.

We use None to represent empty sets of coalternatives and unset branches. Because these can represent different codata types, we rely on polymorphism in the

Declarations

$$
\begin{array}{rcl}
[\![\ \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \quad \big(\mathsf{K}_i : B_i^m \to \mathsf{T}\ X^n\big)^j \ ]\!] & \triangleq & \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \quad \big(\mathsf{K}_i : B_i^m \to \mathsf{T}\ X^n\big)^j \\
[\![\ \mathbf{codata}\ \mathsf{U}_h\ X^n\ \mathbf{where} \quad \big(\mathsf{H}_i : \mathsf{U}_h\ X^n \to B_i\big)^j \ ]\!] & \triangleq & \mathbf{data}\ \mathsf{T}\ X^n\ \mathbf{where} \quad \mathsf{K}_h : (\mathsf{Option}\ (\mathbf{Lazy}\ B_i))^j \to \mathsf{T}\ X^n
\end{array}
$$

Terms

$$
\begin{array}{rcl}
[\![x]\!]_d & \triangleq & x \\
[\![\mathbf{fix}\, x\ \mathbf{in}\ t]\!]_d & \triangleq & \mathbf{fix}\, x\ \mathbf{in}\ [\![t]\!]_d \\
[\![\mathsf{K}\ t^n]\!]_d & \triangleq & \mathsf{K}\ [\![t]\!]_d^n \\
[\![\mathbf{case}\ t\ \{(p \to u)^n\}]\!]_d & \triangleq & \mathbf{case}\ [\![t]\!]_d\ \{(p \to [\![u]\!]_d)^n\} \\
[\![\mathbf{cocase}\ ([\cdot]\ e)\ t]\!]_d & \triangleq & [\![t]\!]_d\ [\![e]\!]_d \\
[\![\mathbf{cocase}\ (\mathsf{H}_i\ [\cdot])\ t]\!]_d & \triangleq & \mathbf{case}\ [\![t]\!]_d \left\{ \begin{array}{rcl} \mathsf{None} & \to & \mathbf{fail} \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ \mathsf{None}_i\ \ldots\ x_n) & \to & \mathbf{fail} \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ (\mathsf{Some}\ y)_i\ \ldots\ x_n) & \to & \mathbf{force}\ y \end{array} \right\} \\
 & & \quad \text{where } \mathsf{H}_i \mapsto_d K \\
[\![\{[\cdot]\ x \to t\}]\!]_d & \triangleq & \lambda x.\, [\![t]\!]_d \\
[\![\{\mathsf{H}_i\ [\cdot] \to t\ ,\ \_ \to u\}]\!]_d & \triangleq & \mathbf{case}\ [\![u]\!]_d \left\{ \begin{array}{c} \mathsf{None} \to \\ \mathsf{Some}\ (\mathsf{K}\ \mathsf{None}_0\ \ldots\ (\mathsf{Some}\ (\mathbf{thk}\ [\![t]\!]_d))_i\ \ldots\ \mathsf{None}_n) \\ \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ x_i\ \ldots\ x_n) \to \\ \mathsf{Some}\ (\mathsf{K}\ x_0\ \ldots\ (\mathsf{Some}\ (\mathbf{thk}\ [\![t]\!]_d))_i\ \ldots\ x_n) \end{array} \right\} \\
 & & \quad \text{where } \mathsf{H}_i \mapsto_d K \\
[\![\{\}]\!]_d & \triangleq & \mathsf{None}
\end{array}
$$

Figure 16. Translation from λ_F^{cop} to $\lambda_{\mathcal{V}}^{pat}$

target language so that the program type checks. In an untyped translation, this is not a problem.

4.3.2 Call-by-value. Terms do not represent computations in call-by-value so we require more machinery to prevent computations from running before a destructor is applied, see Figure 16. If we just put the translated branches into our generated data type as we did with call-by-name then the computation could infinitely loop even if not demanded.

We can control when computations happen with λ's because they have the property that they only run the body upon observing an applicative context. For simplicity, we use the following macros to transform types and terms into thunked

types and terms.

$$\begin{aligned} \textbf{Lazy}\ A &\triangleq_{\text{type}} () \to A \\ \textbf{force}\ e &\triangleq e\ () \\ \textbf{thk}\ e &\triangleq \lambda\_.\, e \end{aligned}$$

We delay computations with **thk** and evaluate them on-demand with **force**. The type changes to a function $() \to A$ as seen in Chapter II. In Chapter V, we show how to reimpliment these macros with mutation as an optimization.

Since the arguments of our generated constructor must represent unevaluated branches, we must wrap their types with our lazy type. To visualize the change, let us again look at the running example:

$$\begin{array}{l} \textbf{codata}\ \mathsf{Stream}\ A\ \textbf{where} \\ \quad \mathsf{Head} : \mathsf{Stream}\ A \to A \\ \quad \mathsf{Tail} : \mathsf{Stream}\ A \to \mathsf{Stream}\ A \end{array}$$

$$\begin{array}{l} \textbf{data}\ \mathsf{Stream}\ A\ \textbf{where} \\ \quad \mathsf{MkS} : \mathsf{Option}\ (\textbf{Lazy}\ A), \mathsf{Option}\ (\textbf{Lazy}\ (\mathsf{Stream}\ A)) \to \mathsf{Stream}\ A \end{array}$$

The term translation also needs to produce and eliminate these delayed computations. We thunk terms before we store elements in the generated

constructor and force them only when we apply a destructor.

$$\textbf{cocase}\ (\text{Head}\ [\cdot])\ \left(\textbf{fix}\, x\ \textbf{in}\ \left\{\begin{array}{c}\text{Head}\ [\cdot] \to 0\\ \_ \to \left\{\begin{array}{c}\text{Tail}\ [\cdot] \to x\\ \_ \to \{\}\end{array}\right\}\end{array}\right\}\right)$$

$$\textbf{case}\ \left(\textbf{fix}\, s\ \textbf{in}\ \left(\begin{array}{l}\textbf{case}\ \left(\begin{array}{l}\textbf{case}\ \text{None}\\ \quad \text{None} \to \text{Some}\ (\text{MkS}\ \text{None}\ (\text{Some}\ (\textbf{thk}\ s)))\\ \quad \text{Some}\ (\text{MkS}\ x_0\ x_1) \to \text{Some}\ (\text{MkS}\ x_0\ (\text{Some}\ (\textbf{thk}\ s)))\end{array}\right)\\ \quad \text{None} \to \text{Some}\ (\text{MkS}\ (\text{Some}\ (\textbf{thk}\ 0))\ \text{None})\\ \quad \text{Some}\ (\text{MkS}\ x_0\ x_1) \to \text{Some}\ (\text{MkS}\ (\text{Some}\ (\textbf{thk}\ 0))\ x_1)\end{array}\right)\right)$$

$$\begin{array}{l}\text{None} \to \textbf{fail}\\ \text{Some}\ (\text{MkS}\ \text{None}\ x_1) \to \textbf{fail}\\ \text{Some}\ (\text{MkS}\ (\text{Some}\ x_0)\ x_1) \to \textbf{force}\ x_0\end{array}$$

4.3.3 Correctness. The whole compilation pipeline is denoted as $(\!|-|\!)$; it is the composition of flattening and codata elimination, or $[\![-]\!] \circ (\twoheadrightarrow_F)$. To guarantee our pipeline preserves the meaning of programs, we show it is sound.

Theorem 6 (Soundess)**.** *Forall $t, t' \in \lambda^{cop}$, if $t \mapsto t'$ then for both call-by-name and call-by-value translation $(\!|t|\!) \mapsto\!\!\!\!\twoheadrightarrow (\!|t'|\!)$ under their respective evaluation strategy.*

CHAPTER V

IMPLEMENTATIONS

After focusing on codata from a theoretical point of view, we examine codata in real world compilers and applications. The translation of λ^{cop} given in Chapter IV has been implemented both as a prototype compiler with multiple backends and as a language extension for Haskell. The compiler's source code is an ASCII version of λ^{cop} just as it was described in Chapter III. We can compile it to Racket, Ocaml and Haskell demonstrating that our compilation technique works in an untyped call-by-value, typed call-by-value, and call-by-name settings, respectively. As a language extension, we add codata and copattern matching to the Haskell language.

In the Section 5.1, we will describe the hurdles and optimizations we discovered while implementing the different backends for our prototype compiler. In the Section 5.2, we look at the codata extension to Haskell inspecting its benefits, performance, and giving an example of combining codata with the `IO` monad.

5.1 λ^{cop} Prototype Compiler

Our prototype compiler takes a λ^{cop} program and generates a program in one of our backend languages. Example output of the compiler for each backend can be found in Appendix A as well as an example of the ASCII source code. For each target language, we had to alter the translation slightly. In general, we had issues with code duplication in flattening and preserving extensionality of codata after compilation. We were able to add a sharing optimization to the generated code that results in a dramatic performance gain for some applications. The different target languages also allowed us to simplify the translation in the case of flattening patterns, handling failures, and handling missing coalternatives.

5.1.1 Code De-duplication. The flattening rules of the operational semantics duplicate sub-terms when copatterns are unnested. And since each copattern needs to be unnested during the compilation process, there is a large amount of code duplication with a naïve implementation.

$$\{q[[\cdot]\ p] \to t\ ,\ coalt^n\} \ \mapsto_F \ \left\{ \begin{array}{lcl} [\cdot]\ p & \to & \left\{ \begin{array}{lcl} q & \to & t \\ \_ & \to & \{coalt^n\}\ p \end{array} \right\} \\ \_ & \to & \{coalt^n\} \end{array} \right\}$$

$$\{q[\mathsf{H}_i\ [\cdot]] \to t\ ,\ coalt^n\} \ \mapsto_F \ \left\{ \begin{array}{lcl} \mathsf{H}_i\ [\cdot] & \to & \left\{ \begin{array}{lcl} q & \to & t \\ \_ & \to & \mathsf{H}_i\ \{coalt^n\} \end{array} \right\} \\ \_ & \to & \{coalt^n\} \end{array} \right\}$$

In the rules that flatten copatterns above, we see that the rest of the coalternatives in the two default cases is duplicated.

To resolve this, we simply insert let expressions in our implementation. The code becomes the following.

$$\{q[[\cdot]\ p] \to t\ ,\ coalt^n\} \ \mapsto_F \ \begin{array}{l} \textbf{let}\ x = \{coalt^n\}\ \textbf{in} \\ \left\{ \begin{array}{lcl} [\cdot]\ p & \to & \left\{ \begin{array}{lcl} q & \to & t \\ \_ & \to & x\ p \end{array} \right\} \\ \_ & \to & x \end{array} \right\} \end{array}$$

$$\{q[\mathsf{H}_i\ [\cdot]] \to t\ ,\ coalt^n\} \ \mapsto_F \ \begin{array}{l} \textbf{let}\ x = \{coalt^n\}\ \textbf{in} \\ \left\{ \begin{array}{lcl} \mathsf{H}_i\ [\cdot] & \to & \left\{ \begin{array}{lcl} q & \to & t \\ \_ & \to & \mathsf{H}_i\ x \end{array} \right\} \\ \_ & \to & x \end{array} \right\} \end{array}$$

Post optimization the output code size is now linear with the input program. Unless our target language's compiler can do the common sub-expression elimination, our output program also uses less memory because each duplication of coalternatives corresponds to another generated object for the codata type.

5.1.2 Preserving Extensionality for Compiled Codata. Since the prototype compiler translates λ^{cop} programs into real world programming languages, we can make use of the target language features when observing a λ^{cop} object. When we perform the translation, a codata structure is turned into a data structure in the target language; in addition, the declared destructors are turned into functions in the target language that can operate on the compiled code. For instance, the λ^{cop} term $\{\mathsf{Fst}\ [\cdot] \to 42, \mathsf{Snd}\ [\cdot] \to \mathsf{True}\}$ gets compiled into some structure $x : \mathbb{Z}\ \&\ \mathsf{Bool}$ in our target language along with two functions $\mathsf{Fst} : \mathbb{Z}\ \&\ \mathsf{Bool} \to \mathbb{Z}$ and $\mathsf{Snd} : \mathbb{Z}\ \&\ \mathsf{Bool} \to \mathsf{Bool}$. This allows a programmer in the target language to build up complex observations for codata from the destructors created by compilation.

Since our translation creates data structures in the target representing λ^{cop} codata, a programmer can use a case-expression to inspect the object. Thus we have lost a key property of λ^{cop} by compiling it: we do not have a restriction on the evaluation context in which codata can occur.

The solution to this problem is to use the module system of the target language. Modules allow us to hide the data types and constructors used to create our codata. This prevents programmers from inspecting translated codata with a case-expression preserving the extensionality of our codata. The only way to interact with codata is through the destructor functions which we expose to the users.

5.1.3 Sharing. When we have the same codata appearing in several places of code, it is more efficient to share computations of observations. A small example of this is a program that uses a "with" type.

$$\textbf{let } \mathrm{x} = \left\{ \begin{array}{lcc} \mathsf{Fst}\ [\cdot] & \to & 20+1 \\ \mathsf{Snd}\ [\cdot] & \to & 0 \end{array} \right\} \textbf{ in} \\ \mathsf{Fst}\ \mathrm{x} + \mathsf{Fst}\ \mathrm{x}$$

If we do not share the computation of the branches of x, then we will compute $20 + 1$ twice.

Sharing can potentially result in large performance improvements. The poster child for this is the Fibonacci stream where sharing the sub-computations gives us a linear time algorithm.

$$\mathrm{Fibonacci} = \textbf{fix}\, s \textbf{ in } \left\{ \begin{array}{lcc} \mathsf{Head}\ [\cdot] & \to & 1 \\ \mathsf{Head}\ [\mathsf{Tail}\ [\cdot]] & \to & 1 \\ \mathsf{Tail}\ [\mathsf{Tail}\ [\cdot]] & \to & \mathrm{zipWith}\ (+)\ s\ (\mathsf{Tail}\ s) \end{array} \right\}$$

We see the stream s appears duplicated in the last coalternative.

In our call-by-name backend, we get sharing for free because Haskell is in fact call-by-need. And in our call-by-value backends, we need only make small changes to the translation given in Chapter IV.

Recall that in the call-by-value translation we controlled the computation of the branches of coalternative lists by wrapping the terms in thunks. For soundness thunks are enough, but we can implement the thunks more efficiently if we have mutation in the target language (as we do in both Ocaml and Racket). We can rework the definitions of **thk** and **force** given in the translation so that we have

Language	Program	n	Time(s)
Haskell	fib_with_codata	40	0.004
Haskell	fib_non-sharing	40	10.208
Ocaml	fib_with_codata	40	0.004
Ocaml	fib_non-sharing	40	48.342
Racket	fib_with_codata	40	0.612
Racket	fib_non-sharing	40	2.283

Table 1. fib(40) micro-benchmarks for λ^{cop} backends.

sharing.

$$\textbf{Lazy}\ A \triangleq_{\text{type}} \mathsf{Ref}\ (A + () \to A)$$

$$\begin{array}{lll} & \textbf{case}\ \ !e & \\ \textbf{force}\ e \triangleq & \quad \mathsf{Left}\ v \to v & \\ & \quad \mathsf{Right}\ t \to e := \mathsf{Left}\ (t\ ())\ ;\ \textbf{force}\ e & \end{array}$$

$$\textbf{thk}\ e \triangleq \textbf{ref}\ (\mathsf{Right}\ \lambda\_.\, e)$$

In Table 1, we demonstrate that for each backend the code generated for the Fibonacci stream is the linear time algorithm. This means that we are able to share sub-computations. The figure compares our generated Fibonacci program to a handwritten Fibonacci which does not share.

5.1.4 On Specific Backends. Racket was chosen as a backend to demonstrate the simplicity of our compilation technique. Other work in codata and copatterns relies on strong type systems existing in a proof assistant or a language with mild dependent types. Our compilation to Racket is completely

untyped. Instead of translating the type declarations in a λ^{cop} program to a type declaration in the target, declarations are translated into functions to construct S-expressions and each destructor is translated into function that pattern matches on an S-expression return that destructor's index. For instance, the Stream codata type would store its branches in the S-expression `‘(Stream ,hd ,tl)`, where `hd` and `tl` are thunks representing the computation to be run on observation.

For the typed backends, the Ocaml and Haskell translations were nearly identical to the translations in Chapter IV. We simplified them by making use of each language's record syntax. We declared records which gave us projections instead of having to do extra pattern matches to get a particular branch of a coalternative.

In Haskell, we were also able to avoid checking and constructing optional types as in the translation by making use of Haskell's lazy, polymorphic failure operation called `error`. This, along with not having to force and thunk to get sharing makes the Haskell generated code the simplest of the three backends.

5.2 Copatterns Haskell Language Extension

Our small language λ^{cop} is easy to reason about, but it lacks all of the tools available for more interesting examples using codata, such as effects like `IO`. To obtain this extra power, we extended the Glasgow Haskell Compiler with codata and copattern matching. In addition to having effects, extending Haskell gave us a strong enough type system to encode indexed codata.

5.2.1 Codata with IO. We modified the small server example from Chapter II to make use of `IO`. This allowed us to create a real, executable server defined by codata as we see written in Haskell below.

```
codata Server where
  Get  :: Server -> (String,Server)
  Post :: Server -> String -> Server
```

The destructor `Get` will generate output and return a new server whereas `Post` will be used to update the servers with some String.

Below we defined a server that acts like a stack. We can continually push strings on the stack as we create `Post` messages. Note that we use the ASCII `#` for the copattern $[\cdot]$.

```
stack :: Server
stack =
  let g = {  Get  [# (x:xs)] -> (x,g xs)
          ; [Post    [# xs]] s -> g (s:xs) }
  in { [Post #] s -> g [s] }
```

To connect our server to the real world, we create a simple function that builds contexts for our codata based on command-line input and output.

```
main :: IO ()
main = forever (contextBuilder stack)
  where contextBuilder s =
          getLine
          >>= { # "GET"  ->
                  let (x,s') = Get s in
                    putStrLn x >> return s'
              ; # "POST" ->
                   do l <- getLine
                      return (Post s l) }
          >>= context
```

When we read a string `"GET"` from the console, we apply the `Get` destructor to our server, print that output to the console, and continue with the new state of the server. When we read a string `"POST"` from the console, we wait for another line and upon receiving it we add it to our server through the `Post` destructor. Of course, we could add the type indices to our server codata type to prevent us from using `Get` when the list of posts is empty, but to demonstrate connecting our codata to `IO` this is satisfactory.

5.2.2 Performance. We also did a comparison of the GHC language extension to lazy lists, Table 2. Lists are the standard tool for creating streams and our comparison demonstrates that codata streams in the extension achieves the same performance. These benchmarks include the common codata example programs of Fibonacci and prime number streams.

Program	Implementation	n	Time(s)
fib	codata	1000	0.0355
fib	codata	10000	3.116
fib	codata	100000	329.979
fib	list	1000	0.030
fib	list	10000	3.041
fib	list	100000	317.293
prime	codata	1000	0.064
prime	codata	10000	11.772
prime	list	1000	0.026
prime	list	10000	3.030

Table 2. Micro-benchmarks for GHC implementation.

CHAPTER VI

APPLICATIONS

The question remains of why we would want have codata in our language rather than records or object from object-oriented languages. To answer this question and also to present more examples of codata, we will focus on programming applications where codata's features make it a natural abstraction to use. In addition to programming, we will show that codata can beneficial for reasoning because of its extensionality laws.

6.1 Programming

Choosing to represent a problem with codata can help with writing observation based programs like infinite structures, processes, and applications that constrain the users. These types of programs share the common idea that when requests are made of an object they trigger computation. We will show an example of transforming codata to build new codata. Then we demonstrate how indexed codata can help us encode the notion of resource sharing and for security by controlling access to operations and information.

6.1.1 Transforming Codata. In his popular paper "Why functional programming matters", Hughes demonstrates with a number of useful examples both that function programming allows us to construct complex programs from smaller ones and that laziness gives us flexibility when building programs [9]. These examples include calculating the nth approximation of square-roots, derivatives, and integrals on-demand. The on-demand aspect of these programs is what makes them fit for codata because on-demand can also be read as "an action to perform when an observation is applied". Codata also has a distinct advantage over

constructing objects with laziness: it does not depend on a particular evaluation strategy.

To show that codata is rich enough to create complex programs we will consider the example of constructing a stream of prime numbers. Like in Hughes, we will break this problem into smaller parts.

A good starting point when creating a stream of prime numbers is a stream of ascending numbers. We define a function "countUp" which takes an integer and returns an infinite stream counting up from that number.

$$\begin{aligned} \text{countUp} &: \mathbb{Z} \to \mathsf{Stream}\ \mathbb{Z} \\ \text{countUp} &= \left\{ \begin{array}{lcl} \mathsf{Head}\ [[\cdot]\ x] & \to & x \\ \mathsf{Tail}\ [[\cdot]\ x] & \to & \text{countUp}\ (x+1) \end{array} \right\} \end{aligned}$$

The stream is built by corecursion on its integer argument. Instead of recursing on a smaller input till a base case, we corecurse on a larger input to infinity. With "countUp" we can easily construct the stream of positive integers as a function call.

$$\begin{aligned} \text{nats} &: \mathsf{Stream}\ \mathbb{Z} \\ \text{nats} &= \text{countUp}\ 1 \end{aligned}$$

Though we have bound some codata to the identifier "nats", none of the elements of the codata have been computed yet. In call-by-name, we have only created a pointer to a thunk, and in call-by-value, we have only created a structure that houses two thunks for the Head and Tail.

To get the prime numbers from a stream of positive integers, we need to be able to remove elements from a stream. We can define a higher-order function operating on streams called "filter", which after taking a predicate and a stream

returns a new stream containing only the elements of the stream that satisfy the predicate.

$$
\begin{array}{lcl}
\text{filter} & : & (A \to \mathsf{Bool}) \to \mathsf{Stream}\ A \to \mathsf{Stream}\ A \\
\text{filter}\ p\ s & = & \begin{array}{l}
\quad \mathbf{let}\ h = \mathsf{Head}\ s\ \mathbf{in} \\
\quad\ \mathbf{if}\ p\ h \\
\quad\ \mathbf{then} \left\{ \begin{array}{lcl} \mathsf{Head}\ [\cdot] & \to & h \\ \mathsf{Tail}\ [\cdot] & \to & \text{filter}\ p\ (\mathsf{Tail}\ s) \end{array} \right\} \\
\quad\ \mathbf{else}\ \text{filter}\ p\ (\mathsf{Tail}\ s)
\end{array}
\end{array}
$$

If the head of the current stream satisfies the predicate than we include it in the output stream. The output stream is constructed by corecursively calling "filter" on the tail of the stream. In functional programming languages, "filter" is a common higher-order function used for lists and other data structures. The stream version looks and works similar to its data counterparts. Since we are using codata, the elements of the stream are not computed when we apply "filter" to arguments. For example, the call "filter (> 0) nats" simply construct a new stream that can compute the filtered stream when a destructor is applied.

Now that we can remove elements from streams, we need to specify how to remove elements that are factors of earlier elements in the stream, which is the essential part of the prime number stream.

$$
\begin{array}{lcl}
\text{sift} & : & \mathsf{Stream}\ \mathbb{Z} \to \mathsf{Stream}\ \mathbb{Z} \\
\text{sift}\ s & = & \left\{ \begin{array}{lcl} \mathsf{Head}\ [\cdot] & \to & \mathsf{Head}\ s \\ \mathsf{Tail}\ [\cdot] & \to & \text{sift}\ (\text{filter}\ (\lambda x.\, \text{mod}\ x\ (\mathsf{Head}\ s))\ (\mathsf{Tail}\ s)) \end{array} \right\}
\end{array}
$$

"sift", when given a stream, constructs a new stream such that when we destruct with Tail, we compute the next stream that has filtered out all of the multiples of the current Head. This is done corecursively for each new state of the stream as we apply Tail destructors.

Finally, with our "sift" helper function we can succinctly describe a stream of prime numbers as the following.

$$\begin{aligned} \text{primes} &: \mathsf{Stream}\ \mathbb{Z} \\ \text{primes} &= \text{sift}\ (\mathsf{Tail}\ \text{nats}) \end{aligned}$$

If we want the second element of "primes" for instance, we start with the stream "countUp 1". Getting the Tail of that stream, we have "countUp 2". "sift" turns this into a stream where the Head is 2 and the Tail is a stream where all of the elements are not divisible by 2, that is, $4, 6, 8, \ldots$ would no longer be in the stream.

We have defined the stream containing the prime numbers by transforming simpler codata. And since we were using codata, all of the serious work of computing the elements is only done when we destruct with Head and Tail. This is strong motivation for codata in a call-by-value setting, but in a call-by-name language we would get all of the same properties by default.

6.1.2 Programming with Indexed Codata. The call-by-name motivation for codata is much stronger when we consider how easily we can encode client-constraining invariants with indexed codata. We will give two examples of applications with these constraints that are fair scheduling and access control. And though these encodings are also possible with indexed data, they do not appear as elegant as their codata counterparts. This is analogous to the fact that all data can be encoded with Church encodings, but using data makes the code shorter, easier

to read, and the programmer does not need to worry about how to Church encode their data.

6.1.2.1 Resource Sharing. A small example of using codata to describe resource sharing is a fair stream. That is a combination of two streams with the invariant that we view the elements of each sub-stream at the same rate. With this description, a fair stream can be seen as a primitive scheduler giving equal time to two processes. We represent this codata with the following indexed declaration where the indices are underlined.

codata FairStream X $\underline{L}$ $\underline{R}$ **where**

$$\begin{array}{lcl} \mathsf{Left} & : & \mathsf{FairStream}\ X\ \underline{\mathsf{Unread}}\ \underline{R} \to X\ \&\ \mathsf{FairStream}\ X\ \underline{\mathsf{Read}}\ \underline{R} \\ \mathsf{Right} & : & \mathsf{FairStream}\ X\ \underline{L}\ \underline{\mathsf{Unread}} \to X\ \&\ \mathsf{FairStream}\ X\ \underline{L}\ \underline{\mathsf{Read}} \\ \mathsf{Next} & : & \mathsf{FairStream}\ X\ \underline{\mathsf{Read}}\ \underline{\mathsf{Read}} \to \mathsf{FairStream}\ X\ \underline{\mathsf{Unread}}\ \underline{\mathsf{Unread}} \end{array}$$

The type variable X is the type of elements of the two streams. The variables $\underline{L}$ and $\underline{R}$ range over the indices representing the state of the left and right stream, respectively. For fair streams, the indices can take the values $\underline{\mathsf{Read}}$ and $\underline{\mathsf{Unread}}$. As specified in the definition of the destructor, reading from the left stream can only be done when the left stream tag is $\underline{\mathsf{Unread}}$. The result of the projection is the value in the left of the stream and a new stream where the left stream tag is set to $\underline{\mathsf{Read}}$. In order to read from the left side a second time, the client must first apply the destructor Next which requires that the right stream has been viewed as well.

We construct an instance of a fair stream below that represents the integers.

$$\text{ints} = \left(\mathbf{fix}\, s\ \mathbf{in} \left\{ \begin{array}{rcl} \mathsf{Fst}\ [\mathsf{Left}\ [[\cdot]\ x]] & \to & -x \\ \mathsf{Snd}\ [\mathsf{Left}\ [[\cdot]\ x]] & \to & s\ x \\ \mathsf{Fst}\ [\mathsf{Right}\ [[\cdot]\ x]] & \to & x \\ \mathsf{Snd}\ [\mathsf{Right}\ [[\cdot]\ x]] & \to & s\ x \\ \mathsf{Next}\ [[\cdot]\ x] & \to & s\ (x+1) \end{array} \right\} \right)\ 1$$

Using a magnitude as the state, the left stream returns a negative view of the state whereas the right stream returns the positive view. Observing the next section of the state means we increase the magnitude of the state by 1.

When applying destructors to "ints", the client must satisfy the invariant defined in the codata declaration. The client has the freedom to read from each side in any order, but it must ask for the next section of the fair stream before it can read from the same side again. When copattern matching on indexed codata observations that do not satisfy the invariant will not compile. For example, the following program would result in a type error

$$\mathsf{Left}\ (\mathsf{Snd}\ (\mathsf{Left}\ \text{ints}))$$

while the following two interactions pass the type checker.

$$\mathsf{Left}\ (\mathsf{Snd}\ (\mathsf{Right}\ \text{ints}))$$
$$\mathsf{Right}\ (\mathsf{Snd}\ (\mathsf{Left}\ \text{ints}))$$

6.1.2.2 Access Control. Indexed codata also provides a secure way to encode access control. As an example let us consider a filesystem where we can create, delete, and read files. We have a notion of root and unprivileged user where the root user has control over which files are in the system while the unprivileged user can only view files. We would also like unprivileged user to be able to run root destructors given a password.

We define the filesystem as codata parameterized by the users index, where we have the user indices of Root and Unprivileged. The index variable $\underline{U}$ stands for either root or unprivileged users.

codata FS $\underline{U}$ **where**

Promote	:	FS Unprivileged $\to$ String $\to$ FS Unprivileged + FS Root
Demote	:	FS Root $\to$ FS Unprivileged
CreateFile	:	FS Root $\to$ String $\to$ String $\to$ Ref & FS Root
DeleteFile	:	FS Root $\to$ Ref $\to$ FS Root
ReadFile	:	FS $\underline{U}$ $\to$ Ref $\to$ Option String

The Promote destructor allows an Unprivileged instance of the codata to become a Root instance if it provides the correct password (represented with a String). Once we have a Root instance we can create files, delete files, and returned to an Unprivileged instance. Both root and unprivileged users can read files given a pointer to one.

We need more type declarations before we can implement an instance of the filesytem. We use the following codata type for references or pointers to files.

$$\textbf{codata}\ \mathsf{Ref}\ \textbf{where}$$

$$\mathsf{Reference} : \mathsf{Ref} \to \mathsf{String}$$

The String is just the name of a file. We also require some internal state to hold the names and contents of files.

$$\textbf{type}\ \mathsf{State} = \mathsf{List}\ (\mathsf{Ref} \times \mathsf{String})$$

We represent the internal state as a list that pairs reference to a file and its contents. We also have functions "remove : Ref → State → State" and "lookup : Ref → State → Option String" to modify our state depending on the reference.

We define a file system with two mutually recursive parts: one with the index Unprivileged and the other with the index Root. They take the current state as an argument, but since we are defining the file system as codata a client can only access the state in the ways defined by the destructors.

$$\begin{aligned}
&\text{unprivileged} \quad : \quad \mathsf{State} \to \mathsf{FS}\ \underline{\mathsf{Unprivileged}} \\
&\text{unprivileged} \quad = \quad \left\{ \begin{array}{l}
[\mathsf{Promote}\ [[\cdot]\ st]]\ p \to \\
\qquad \textbf{case}\ (\text{“password”} = p) \\
\qquad\quad \mathsf{True} \to \text{root}\ st \\
\qquad\quad \mathsf{False} \to \text{unprivileged}\ st \\
[\mathsf{ReadFile}\ [[\cdot]\ st]]\ ref \to \text{lookup}\ (\mathsf{Ref}\ ref)\ st
\end{array} \right\}
\end{aligned}$$

The "unprivileged" function only needs to consider two observations because these are all that is possible given the index Unprivileged. The Promote destructor will match its input against the hard-coded password and return either an Unprivileged or Root indexed filesytem depending on if the password matches. ReadFile simply does a lookup for the file requested.

$$
\begin{array}{lcl}
\text{root} & : & \mathsf{State} \to \mathsf{FS}\ \underline{\mathsf{Root}} \\
\text{root} & = & \left\{ \begin{array}{l}
\mathsf{Demote}\ [[\cdot]\ st] \to \text{unprivileged}\ st \\
[\mathsf{CreateFile}\ [[\cdot]\ st]]\ name\ contents \to \\
\qquad \textbf{let}\ ref = \{\mathsf{Ref}\ [\cdot] \to name\}\ \textbf{in} \\
\qquad\quad \left\{ \begin{array}{lll}
\mathsf{Fst}\ [\cdot] & \to & ref \\
\mathsf{Snd}\ [\cdot] & \to & \text{root}\ (\mathsf{Cons}\ (ref, contents)\ st)
\end{array} \right\} \\
[\mathsf{DeleteFile}\ [\cdot]\ st]\ ref \to \\
\qquad \text{root}\ (\text{remove}\ (\mathsf{Ref}\ ref)\ st) \\
[\mathsf{ReadFile}\ [[\cdot]\ st]]\ ref \to \text{lookup}\ (\mathsf{Ref}\ ref)\ st
\end{array} \right\}
\end{array}
$$

For the section of the filesystem where we have Root observations available we have more to do. Most notably, if we apply the Demote destructor then we return to a filesystem with a Unprivileged index limiting our possible actions again. The CreateFile and DeleteFile just manipulate the internal state.

Finally we can construct a starting filesystem by giving an initial state.

$$
\begin{array}{lcl}
\text{filesystem} & : & \mathsf{FS}\ \underline{\mathsf{Unprivileged}} \\
\text{filesystem} & = & \text{unprivileged}\ \mathsf{Nil}
\end{array}
$$

This instance completely hides the state from both root and unprivileged users. And because we are using codata, the only way to access this state is with the declared destructors.

Below is an example of a client interaction with this filesystem instance. The client tries to promote itself to a root user with a password. If the promotion succeeds, then the client creates a file before demoting itself and reading the file from an unprivileged view.

```
case (Promote filesystem "pass")
  Left fs' → None
  Right fs' →
      let m = CreateFile fs' "file" "text" in
        ReadFile (Demote (Snd m)) (Fst m)
```

The ability to use certain destructors is maintained by the type indices. For example, if we were to use CreateFile on the unprivileged filesystem bound in the Left case, then we would get a type error since we did not have a codata instances with the Root index. Indexed codata makes it easy for us to define these invariants; this is a benefit of codata that is an improvement for both call-by-name and call-by-value strategies.

6.2 Reasoning

Adding codata to a language gives us terms that satisfy the extensional (η) laws. This allows us to improve our ability to reason about our programs. We will focus on Haskell because the advantage is more clear cut.

The η-law for functions is commonly seen in definitions of the λ-calculus: if M is a function, then $M \equiv_\eta \lambda x.\, M\ x$. However as we will see, η does not hold

for functions in Haskell. To understand why this is important, let us consider a commonly used typeclass in Haskell: the monad.

$$\begin{aligned}&\textbf{class}\ \mathsf{Monad}\ M\ \textbf{where}\\ &\quad \text{return} : A \to M\ A\\ &\quad \text{bind} : M\ A \to (A \to M\ B) \to M\ B\end{aligned}$$

As clients of some monad implementation, we expect that it was constructed in a sensible way. Library writers in Haskell provide this guarantee by showing that their monad instance satisfies the monad laws. This allows clients to rewrite their code while preserving its meaning. One of these monad laws is the left-identity law.

Theorem (Left identity)**.**

$$\begin{aligned}\forall x : A,\ \ f : \mathsf{Monad}\ M \Rightarrow A \to M\ B.\\ \text{bind}\ (\text{return}\ x)\ f = f\ x.\end{aligned}$$

To show why extensionality is important for proving this monad law, we will use the state monad as an example. The state monad allows us to create a computation that manipulates state as a side-effect. We can implement the monad with Haskell's function type.

$$\textbf{type}\ \mathsf{State}\ S\ A = S \to S \times A$$

$$\begin{aligned}&\textbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ S)\ \textbf{where}\\ &\quad \text{return}\ x = \lambda s.\, \mathsf{Pair}\ s\ x\\ &\quad \text{bind}\ m\ f = \lambda s.\, \textbf{case}\ (m\ s)\ \{\mathsf{Pair}\ s'\ x \to f\ x\ s'\}\end{aligned}$$

And as a Haskell library writer would, we construct a proof the left-identity law for our implementation.

Proof.

Assume,

$$\text{bind (return } x) \ f.$$

By definition of return,

$$\text{bind } (\lambda\, s.\, (s, x)) \ f.$$

By definition of bind,

$$\lambda\, s.\, \textbf{case } ((\lambda\, s'.\, (s', x)) \ s) \ \{(s', x') \to f \ x' \ s'\}.$$

By β,

$$\lambda\, s.\, \textbf{case } (s, x) \ \{(s', x') \to f \ x' \ s'\}.$$

By pattern match,

$$\lambda\, s.\, f \ x \ s.$$

By η,

$$f \ x.$$

□

We have a problem though! The last step of this proof is incorrect because we do not actually have an η-law for functions in Haskell. A counter example is if f is $\bot$.

$$\lambda\, s.\, \bot \ x \ s \not\equiv \bot \ x$$

These two are not equivalent in Haskell because the following context distinguishes them.

$$\textbf{case}\ (\lambda\, s.\, \bot\ x\ s)\ \{y \to 0\} \not\equiv \textbf{case}\ (\bot\ x)\ \{y \to 0\}$$

The left case will evaluate to 0 whereas the right will loop forever.

Codata can save us here because we cannot construct the context which distinguishes these two cases. We start by defining the following special codata function type.

$$\textbf{codata}\ A \rightsquigarrow B\ \textbf{where}$$

$$\mathsf{Ap} : A \rightsquigarrow B \to A \to B$$

With this codata type, we have the following η-law.

$$g : A \rightsquigarrow B \equiv_\eta \{\mathsf{Ap}\ [\cdot]\ s \to \mathsf{Ap}\ g\ s\}$$

This holds because a case expression around a term of type $A \rightsquigarrow B$ will result in a type error.

We now redefine our type State and monad instance.

$$\textbf{type}\ \mathsf{State}\ S\ A = S \rightsquigarrow S \times A$$

$\textbf{instance}\ \mathsf{Monad}\ (\mathsf{State}\ S)\ \textbf{where}$

$\quad \text{return}\ x = \{\mathsf{Ap}\ [\cdot]\ s \to \mathsf{Pair}\ s\ x\}$

$\quad \text{bind}\ m\ f = \{\mathsf{Ap}\ [\cdot]\ s \to \textbf{case}\ (\mathsf{Ap}\ m\ s)\ \{\mathsf{Pair}\ s'\ x \to \mathsf{Ap}\ (f\ x)\ s'\}\}$

Now our proof of the left-identity law holds because η holds for $A \rightsquigarrow B$.

Proof.

Assume,

bind (return x) f.

By definition of return,

bind $\{\mathsf{Ap}\ [\cdot]\ s \to \mathsf{Pair}\ s\ x\}\ f$.

By definition of bind,

$\{\mathsf{Ap}\ [\cdot]\ s \to \mathbf{case}\ (\mathsf{Ap}\ \{\mathsf{Ap}\ [\cdot]\ s' \to \mathsf{Pair}\ s'\ x\}\ s)\ \{\mathsf{Pair}\ s'\ x' \to \mathsf{Ap}\ (f\ x')\ s'\}\}$

By copattern match,

$\{\mathsf{Ap}\ [\cdot]\ s \to \mathbf{case}\ (\mathsf{Pair}\ s\ x)\ \{\mathsf{Pair}\ s'\ x' \to \mathsf{Ap}\ (f\ x')\ s'\}\}$

By pattern match,

$\{\mathsf{Ap}\ [\cdot]\ s \to \mathsf{Ap}\ (f\ x)\ s\}$.

By η,

$f\ x$.

□

Haskell programmers were satisfied with the fast and loose reasoning of the left-identity law for the state monad before. With codata, we have a more correct proof and it is not any more complex than the proof with the built-in function type. So if we need to prove a property that requires η, we can build our structure with codata instead of data.

CHAPTER VII

DISCUSSION

7.1 Comparison with other work

Hagino introduced codata types in his seminal paper [8]. His work includes a dual construct to case expressions which he called "merge". Merge expressions take a list of destructor/term pairs and evaluate only when a matching destructor is applied to them. He notes the asymmetry between case and merge expressions; case expressions take a term as an argument and a list of alternatives whereas merge expressions take only a list of coalternatives. He proposes a modification of merge, "merge′", that pairs a term with a list of coalternatives. His solution does not directly align with our approach because we match coalternatives against observable contexts. Furthermore, Hagino's copatterns are always flat.

Zeilberger looks at the question of duality of connectives from a logical point of view [15]. He discusses the duality between the connectives $A \oplus B$ and $A \otimes B$ used to verify and the connectives $A \,\&\, B$ and $A \parr B$ used to refute. His logical system demonstrated many of the features found in λ^{cop} including nested (co)pattern matching, non-termination, and recursive types.

Downen and Ariola present a sequent calculus perspective of codata and copatterns [6]. In a sense, their system $\mu\tilde{\mu}$ generalizes Zeilberger's allowing users to not only define (co)data types like $A \times B$ and $A \,\&\, B$, but also functions types $A \to B$ and subtraction types $A - B$. Downen's PhD thesis describes a functional language based on $\mu\tilde{\mu}$, that is, a language without control effects [5]. λ^{cop} can be seen as an extension of this language that adds more flexible (co)patterns.

Abel et al. examine copatterns from a proof assistant perspective [1]. They give an algorithm for determining if a set of coalternatives has completely covered

a type. Later, Abel and Pientka use size types to prove well-founded recursion of coinductive programs written with (co)patterns, thus expanding their utility in a theorem prover [2].

In the same vein of work, Setzer et al. extended nested pattern flattening to flattening coverage-complete (co)patterns [12]. Our work can be seen as a more practically focused approach, that is, we flatten (co)patterns to maintain the order of checking and handling of missing patterns in the same way that Ocaml and Haskell evaluate pattern matches.

Thibodeau extends Levy's call-by-push-value with nested copatterns making a connection between Levy's computation types and codata [13]. He compiles away the nested coverage-complete copatterns into flattened codata. Later, Thibodeau et al. extend the type system of this language to include indexed codata types giving examples including a fair bitstream [14], which our fair stream example from Chapter VI is based on.

Regis-Gianas and Laforgue work on implementing indexed codata types in Ocaml inspired much of this work [11]. Unlike Thibodeau, they completely eliminate codata from their language by using generalized algebraic data types (GADTs) to encode observations. They translate codata type declarations into query and dispatch GADTs.

7.2 Contributions

The source language λ^{cop} which we presented contains both (co)data and nested (co)patterns. Our language has a unique emphasis on the duality of matching. We achieve this through a new construct, the cocase expression, which explicitly matches an observable context and a codata type.

Operationally, we presented a parametric semantics for (co)pattern matching which contains a new flattening technique for copatterns. We also specify a syntax directed technique for eliminating codata into call-by-name and call-by-value languages. We implemented the compilation technique in a prototype compiler and as a Haskell language extension. With our technique it was easy to add sharing as an optimization resulting in the dynamic programming version of Fibonacci.

Finally, we provided more evidence of the importance of codata from a programming and reasoning point of view.

7.3 Future Work

Codata has different properties from data and these have not been fully explored from the point of view of a compiler. Perhaps codata's extensionality laws make available more optimizations in the intermediate language. When transforming codata as we did in the prime number example from Chapter VI, we only composed functions avoiding the serious work of computing the data in our codata structure. Intuitively, this is how loop fusion works in Haskell to avoid constructing unnecessary intermediate structures, that is we would rather compute $map\ (g \circ f)$ instead of $map\ g \circ map\ f$.

We explored compiling codata into data in this thesis, but compiling data into codata is just as feasible. For practical languages, we now know we can compile SmallTalk to ML, but can we also compile ML to SmallTalk.

This thesis pushed the duality of matching further by introducing the cocase expression and giving a strategy parametric semantics. However, we still have some asymmetries in λ^{cop}. What do our translations mean in a language where patterns can contain copatterns in addition to copatterns containing patterns? We know this would require control effects.

In conclusion, this thesis presents more examples of use of codata. Our compilation technique has been shown to work in a number of languages. The most important future work is to get codata in the hands of as many programmers as we can. This will for sure foster more collaboration between the theory and practice of programming languages.

APPENDIX

CODE GENERATION

Here we show the generated output of the following λ^{cop} program for constructing the codata representing an infinite stream of zeroes, then we access the third element.

```
codata Stream a
  { Head : Stream a -> a
  , Tail : Stream a -> Stream a }


Head (Tail (Tail (fix x in
                    { Head # -> 0
                    , Tail # -> x })))
```

We describe the additions in the code generation needed to implement codata starting from the simplest: call-by-name version; to the most work: the untyped call-by-value version.

In general, the real code we generate has one change from the translation given in Chapter IV: the case expressions that were used to get and set fields in our translated codata are pulled into top-level functions. This has the effect of shortening the code and making it more readable.

A.1 Haskell

```
module Main where


import Prelude (Show, IO, error, print, (+))


data Stream a
```

```
  = Stream
  { _Head :: a
  , _Tail :: Stream a
  } deriving Show

set_Head cd br =
  Stream (br) (_Tail (cd))
set_Tail cd br =
  Stream (_Head (cd)) (br)

prog =
  _Head (_Tail (_Tail (let { x =
    set_Head (set_Tail ((error "match fail")) (x)) (0) }
  in
    set_Head (set_Tail ((error "match fail")) (x)) (0))))

main :: IO ()
main = print prog
```

We make use of the record syntax in Haskell to avoid having to write our own accessor helper functions. The only helper functions we needed to generate were the setters which added computation branches. Applying a destructor to observe the `Head` or `Tail` of a stream only requires that we apply the accessors given by the record declaration. To handle unmatched (co)patterns we simply use the lazy and polymorphic `error` function.

A.2 Ocaml

```
open Lazy


type ('a) stream  =  { getHead : 'a lazy_t option;
                       getTail : 'a stream option lazy_t option;  }
exception UnmatchedCopattern
let unmatched = None;;
let setHead ocd br =
  match ocd with
    None ->
      (Some ({ getHead = (Some (br)); getTail = None; }))
  | Some cd ->
      (Some ({ getHead = (Some (br)); getTail = (cd).getTail; }));;
let obsHead ocd =
  match ocd with
    None ->
      (raise UnmatchedCopattern)
  | Some cd ->
      match (cd).getHead with
        None ->
          (raise UnmatchedCopattern)
      | Some br ->
          (force (br));;
let setTail ocd br =
  match ocd with
```

```
    None ->
      (Some ({ getHead = None; getTail = (Some (br)); }))
  | Some cd ->
      (Some ({ getHead = (cd).getHead; getTail = (Some (br)); }));;
let obsTail ocd =
  match ocd with
    None ->
      (raise UnmatchedCopattern)
  | Some cd ->
      match (cd).getTail with
        None ->
          (raise UnmatchedCopattern)
      | Some br ->
          (force (br));;

let prog =
  (obsHead
     ((obsTail
         ((obsTail
             (let rec x = lazy ((setHead (((setTail unmatched)
                                           ((lazy (force (x)))))))
                                 ((lazy (0))))
              in (force (x))))))));;

print_int prog;;
```

```
print_newline ();;
```

Unlike the Haskell implementation we do not have some polymorphic function `error` to represent unmatched (co)patterns. To remedy this, the observation helpers must be able to handle whether or not it is being applied to the empty list of coalternatives and whether the branch being observed exists. To handle empty coalternatives, all codata is assumed to be wrapped in an option type. To handle missing branches, all branches are also wrapped in an option type.

A short coming of this code generation scheme is that if we have a list of coalternatives being used as a function, then it cannot be empty. Since we wrap codata in option type, we will get a type error for trying to apply an option type to a value. This problem does not occur in the Haskell version because `error` (used for empty coalternatives and missing branches) is polymorphic and can be of a function type or data type.

A.3 Racket

```
#lang racket
(require racket/promise)

(define unmatched 'none)
(define getHead
  (lambda (cd)
    (match cd
      [`(Stream ,x ,_)
        x])))
(define setHead
  (lambda (ocd)
```

```
    (lambda (br)
      (match ocd
        [`none
          `(some ,`(Stream ,`(some ,br) ,`(none )))]
        [`(some ,cd)
          `(some ,`(Stream ,`(some ,br) ,(getTail cd)))])))))
(define (obsHead ocd)
  (match ocd
    [`none
      (error "unmatched (co)pattern")]
    [`(some ,cd)
      (match (getHead cd)
          [`none
            (error "unmatched (co)pattern")]
          [`(some ,br)
            (force br)])]))
(define getTail
  (lambda (cd)
    (match cd
      [`(Stream ,_ ,x)
        x])))
(define setTail
  (lambda (ocd)
    (lambda (br)
      (match ocd
```

```
        [‘none
          ‘(some ,‘(Stream ,‘(none ) ,‘(some ,br)))]
        [‘(some ,cd)
          ‘(some ,‘(Stream ,(getHead cd) ,‘(some ,br)))])))))
(define (obsTail ocd)
  (match ocd
    [‘none
      (error "unmatched (co)pattern")]
    [‘(some ,cd)
      (match (getTail cd)
          [‘none
            (error "unmatched (co)pattern")]
          [‘(some ,br)
            (force br)])]))

(define prog
  (obsHead
   (obsTail
    (obsTail (letrec ((x ((setHead ((setTail unmatched) (lazy x)))
                          (lazy 0))))
               ((setHead ((setTail unmatched)
                          (lazy x)))
                (lazy 0)))))))

prog
```

Since we are representing our codata as S-expressions, we need to generate a third helper (on top of the two for the Ocaml code generation) for accessing parts of the S-expressions. After that, the generated code looks the same as the Ocaml code. Racket does not have the same short coming as Ocaml when using the empty list of coalternatives as a function because it is untyped; running the code will result instead just a runtime error, which is the same behavior as the operational semantics for λ^{cop} specifies.

REFERENCES CITED

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, 2013.

[2] Andreas M. Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 185–196, 2013.

[3] Lennart Augustsson. Compiling pattern matching. In *Proceedings Of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[4] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 233–243, 2000.

[5] Paul Downen. *Sequent Calculus: a Logic and a Language for Computation and Duality*. PhD thesis, 2017.

[6] Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Structures for structural recursion. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 127–139, 2015.

[7] M. Dummett. The logical basis of methaphysics. In *The William James Lectures, 1976*. Harvard University Press, Cambridge, Massachusetts, 1991.

[8] Tatsuya Hagino. Codatatypes in ML. *Journal of Symbolic Computation*, pages 629–650, 1989.

[9] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[10] G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

[11] Yann Regis-Gianas and Paul Laforgue. Copattern-matchings and first-class observations in ocaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, 2017.

[12] Anton Setzer, Andreas Abel, Brigitte Pientka, and David Thibodeau. Unnesting of copatterns. In *Rewriting and Typed Lambda Calculi*, pages 31–45, 2014.

[13] David Thibodeau. Programming infinite structures using copatterns. Master's thesis, 2015.

[14] David Thibodeau, Andrew Cave, and Brigitte Pientka. Indexed codata types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 351–363, 2016.

[15] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, pages 66–96, 2008.